Efficient Storage of Versioned Matrices

by

Adam B. Seering

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2011

Author .....
Department of Electrical Engineering and Computer Science
January 15, 2011

Certified by .....
Samuel Madden
Associate Professor
Thesis Supervisor

Accepted by .....
Dr. Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee

Efficient Storage of Versioned Matrices

by

Adam B. Seering

Submitted to the Department of Electrical Engineering and Computer Science
on January 15, 2011, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Versioned-matrix storage is increasingly important in scientific applications. Various computer-based scientific research, from astronomy observations to weather predictions to mechanical finite-element analyses, results in the generation of large matrices that must be stored and retrieved. Such matrices are often versioned; an initial matrix is stored, then a subsequent matrix based on the first is produced, then another subsequent matrix after that. For large databases of matrices, available disk storage can be a substantial constraint. I propose a framework and programming interface for storing such versioned matrices, and consider a variety of intra-matrix and inter-matrix approaches to data storage and compression, taking into account disk-space usage, performance for inserting data, and performance for retrieving data from the database. For inter-matrix "delta" compression, I explore and compare several differencing algorithms, and several means of selecting which arrays are differenced against each other, with the aim of optimizing both disk-space usage and insert and retrieve performance.

This work shows that substantial disk-space savings and performance improvements can be achieved by judicious use of these techniques. In particular, a combination of Lempel-Ziv compression and a proposed form of delta compression, it is possible to both decrease disk usage by a factor of 10 and increase query performance for a factor of two or more, for particular data sets and query workloads. Various other strategies can dramatically improve query performance in particular edge cases; for example, a technique called "chunking", where a matrix is broken up and saved as several files on disk, can cause query runtime to be approximately linear in the amount of data requested rather than the size of the raw matrix on disk.

Thesis Supervisor: Samuel Madden
Title: Associate Professor

Acknowledgments

I would like to thank Professor Madden for the many hours of thought and work that he has put in, making sure that I'm on track and helping me to understand how these sorts of large-matrix systems work and are used. I'd also like to thank Philippe Cudre-Mauroux for the same, particularly for sharing his thoughts about algorithms and approaches to matrix differencing and delta ordering, and Professor Stonebraker for sanity-checking my results and helping me to understand what was really going on in some of my systems. And I would be remiss if I didn't thank my family and close friends for all of their support and chocolate-marshmallow cookies as I worked to pull this thesis together.

Contents

List of Figures

List of Tables

Chapter 1

Background and Motivation

Large scientific data sets are becoming increasingly common and important. Modern simulation techniques and sensor arrays generate vast quantities of information to be stored and processed. Much of this data comes, or can be usefully represented, in matrix form. An image from a weather radar, for example, might be stored as a 2-dimensional pixel grid; or a correlations graph might be stored as a sparse matrix.

In certain applications, these matrices can be quite large. For example, the Large Synoptic Survey Telescope plans to gather around 20 gigabytes of imagery data per night every night, and generate at least twice that much processed data every day based on its observations [19], over the course of many years. The US National Oceanic and Atmospheric Administrations NOMADS archive already stores many terabytes of compressed weather and atmospheric data. With such large data sizes, both performance and storage costs can become constraints. It is therefore important to be able to store matrices in a manner that minimizes disk usage and allows for quick access.

1.1 SciDB

SciDB [16] is a matrix-database system that provides a framework for storing, managing, and working with large arrays. SciDB already makes use of a variety of

traditional schemes, such as run-length encoding and Lempel-Ziv compression, to decrease the disk footprint of individual matrices. It aims to be a general-purpose data store for any type of matrix data.

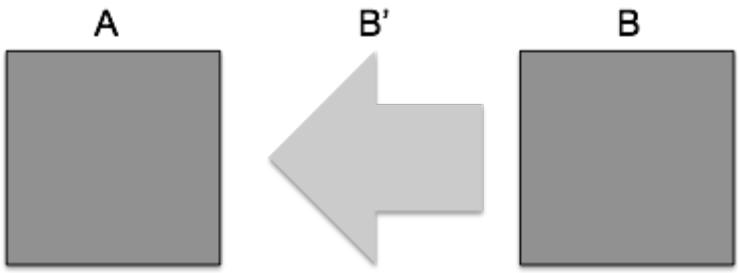
However, these techniques only operate on a single matrix at a time. For certain datasets, storing a collection of arrays as a delta chain or delta tree of versions can both improve performance and greatly decrease disk-space usage.

1.2 Delta Compression

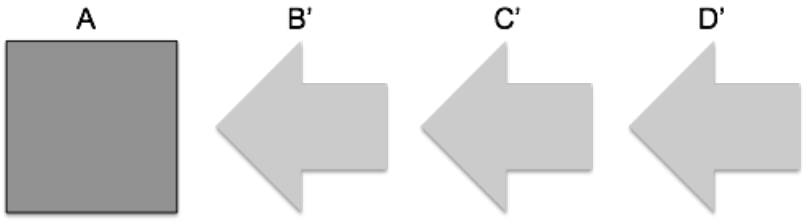
Delta compression in the context of a SciDB-like matrix-storage system is the process of taking a pair of matrices and "subtracting" them using any algorithm that produces some notation of the differences between the two matrices; then storing these differences along with one, rather than both, of the two matrices. Given the difference and the first matrix, one can recover the second matrix. If the two matrices are similar, their difference may be stored using fewer bytes than either matrix requires. A matrix A is said to be delta-compressed against another matrix B if only B and the difference between A and B is stored.

Delta chains, as seen in Figure 1-1, are strings of more than two differenced matrices. For example, given three matrices, the third could be differenced against the second, and the second could be delta-compressed against the first. The third array could also be differenced against the first element in the chain. Doing so would form a delta tree.

Delta compression will yield deltas that are larger or smaller depending on how similar the two arrays in question are. As a result, delta trees can be re-ordered to be more efficient: Take as an example a sparse matrix where each version of the matrix gains one new populated cell. If the versions of the matrix are delta-compressed in chronological order, the result will be one full matrix and a string of single-cell deltas. However, if the versions are delta-compressed in random order, each difference may contain a number of additions, or a number of subtractions, and so may be several times larger.



(a) Array B is differenced against matrix A to create a delta object B', and a delta chain B' to A.



(b) A delta chain is formed with a root of matrix A, and subsequent matrices differenced against each other

Figure 1-1: Matrix Delta, Delta Chain

1.3 Versioned Storage

One typical workflow for a matrix storage system might entail loading in a series of readings from a sensor network, processing each reading through a "cooking" algorithm to identify important features, and querying and comparing each of the resulting arrays. The raw data, and the analyses, are treated as immutable once recorded. This type of workflow typically follows an append-only pattern so that past versions of data can be compared with more-recent versions.

A "versioned storage" model fits this workflow effectively: Each "versioned matrix" has a history of immutable past versions stored in the database as "matrix objects". Periodically, typically when new sensor data has been collected or after a cooking algorithm has completed, the matrix is "committed"; a new immutable version is saved representing the current value of the matrix. If a matrix is to be modified in a new way; for example, to be processed by a new cooking algorithm; a "branch" of the matrix is created that shares a common history with the matrix up

until the point in the matrixs history where the branch was created. This model is conceptually based on that of a version-control system, rather than on a traditional database system.

This access model provides scientists with history- and provenance-tracking features. It also provides some hints to a differencing algorithm about which matrices may be similar: In a large database of matrix objects, objects that are consecutive versions of the same versioned matrix tend to be more similar than array objects picked at random. Versioned storage trees look, at a glance, a great deal like efficient delta trees.

1.4 Motivation

The goal of this research is to develop a system for storing and accessing versioned matrices, and to explore a variety of differencing algorithms against the matrices. Specifically, it aims to do the following:

1. Develop a practical API and library for storing and manipulating versioned matrices

2. Find a set of efficient differencing algorithms that produce a better balance of performance to disk usage than current techniques

3. Explore the creation of delta trees; either determine that simply following versioned-storage trees is in fact the best approach, or illustrate a reasonably-performant algorithm that yields better delta trees

Many database systems do already exist; but few of those are tailored to matrices, and to date, none have carefully studied compression in matrix storage with the aim of both minimizing disk usage and boosting performance. This research aims to achieve both goals.

Chapter 2

Related Work

2.1 SciDB

The SciDB project has studied matrix storage and compression in the past. It has not, however, considered storing matrix differences.

2.2 Binary Differencers

Considerable work has been done towards producing efficient differences of arbitrary objects. An efficient binary-differencing algorithm was proposed by Eugene Myers in 1986 [12]. The algorithm is based on the concept of "edit distance": Given two strings of binary data and a language for expressing operations that can be performed on a string, give the minimal expression in that language that converts the first string to the second.

This algorithm is relatively slow on some data sets. It runs in $O(ND)$ time, where N is the sum of the lengths of the input strings and D is the length of the output string. Also, it's a general algorithm: It is intended to work well on arbitrary data, and as such it makes no optimizing assumptions about the structure of the data it is compressing.

Binary-executable-differencing has been a focus of study for binary differencers, with the aim of reducing the size of the patches that must be downloaded in or-

der to update a piece of software. A small update in a program, even the simple insertion of a few instructions near the beginning of the file, will require pointers throughout the file to be updated as function addresses change and function pointers are adjusted accordingly. A variant of Myers' algorithm was outlined by Colin Percival in 2003 [13]. This algorithm, known as "bsdiff", provides more-compact output with binary files where changes are very sparse, though it will still produce small differences on non-executable binaries.

Various more-specialized algorithms have been proposed as well, such as Google Chrome's Courgette [3], which runs a disassembler on binaries before differencing them to convert function pointers into symbols that need not change. This algorithm does not itself apply to matrices, as arbitrary matrices typically do not contain executable content. However, the experiences of Percival, and of the Google Chrome team, indicate that specialized differencing algorithms can work considerably better than general algorithms that do not take into account the type of the data that they are differencing.

Based on this past work, we plan to consider BSDiff as one starting point for developing matrix-differencing algorithms. We also plan to try to develop a differencing algorithm that takes advantage of the structure inherent in matrices, much like Courgette takes advantage of the structure inherent in software binaries.

2.3 Video Compression

Video compression is another field that has studied matrix compression in some detail. Unfortunately, most development work in the field of video compression has gone into lossy compression, which destroys some amount of data with the aim of decreasing file size. Scientific databases typically expect lossless storage.

The MPEG suite of standards is one of the most common forms of lossy compression. It does lossy compression within frames and between frames. It does offer some ideas that could be extended to lossless compression implementations, however. Perhaps the simplest of these is MPEG-2's use of "I-Frames", periodic

video frames that are not differenced against any other frame [9]. I-Frames are not strictly necessary; the video could simply be a stream of differences. However, it would be virtually impossible to quickly seek around within this stream of video frames; doing so would require starting over at the beginning of the movie (assuming that the first frame is the uncompressed frame in the delta chain) and playing through the video again until the desired location is reached. If I-Frames are present, all that is necessary is to seek to the last I-Frame prior to the desired location, and play forward from that point. The analog for delta chains would be to periodically break the chain by materializing, or un-delta-compressing, matrices in the chain.

MPEG-2 also attempts to compensate for motion between frames. When differencing between frames, it divides up the image being differenced into 16×16-pixel regions, then tries to match each such region to a part of the image being differenced against before performing the actual differencing.

Much of the work in video compression depends on similar work done with image compression. The PNG format [1] is a popular and efficient lossless image compression format that demonstrates the effectiveness of a multi-pass approach: PNG executes an initial filtering pass over the image, using any of a number of filtering algorithms including some that difference adjacent pixels against each other; then executes a compression pass using a generic Lempel-Ziv-based algorithm. The filtering pass, even if unable to reduce the image size, often creates enough redundancy in the image that the Lempel-Ziv compression pass is able to produce a much smaller output file.

Unfortunately, all of these algorithms are limited in that they support a fixed number of dimensions. Video compression stores a three-dimensional matrix (two dimensions per image and one in time); image compression stores a two-dimensional matrix. Additionally, common implementations of these algorithms tend to assume that this data uses 8-bit unsigned integers for cell values, because 8-bit greyscale (or 8 bits per color channel, with color images) is a common standard for consumer graphics. Some implementations of some of these algorithms support 16-bit

unsigned integers; few if any support 32-bit or larger integers, or any other data formats.

We will explore the MPEG-2 idea of trying to match up regions in images that may have shifted relative to each other. Additionally, we will consider periodically materializing matrices in an I-Frame-like manner, and combining differencing and traditional compression methods in ways similar to the techniques used by the PNG image format.

2.4 Versioning

Versioning has been known by many names and been implemented in many forms. One such form is that of version-control systems intended for the tracking of software source code, such as Subversion or Git.

Git in particular is often cited as being faster and more disk-efficient than other similar version-control systems. Significant amounts have been written about its data model [2]: Git stores a version tree and a delta tree, but the two are managed by different software layers and need not relate to each other at all. In order to build an efficient delta tree, Git considers a variety of file characteristics, such as file size and type, in addition to files' relationship in the version tree. It then sorts files by similarity, and differences each file with several of its nearest neighbors to try to find the optimal match.

Git also ensures that its differences will be read quickly by storing consecutive differences in the same file on disk. This way, if several consecutive differences must be read, they will likely be stored consecutively on disk, eliminating the need for additional disk seeks. Additionally, if a client requests these differences over the network, it will receive several (one files' worth) at once, thereby cutting down on the number of required network round-trips.

The concept of taking periodic commits, or snapshots, of a working data set in order to work with historic versions has been known to the databases community

for some time under a different name: Snapshot isolation. Snapshot isolation is a process where a transaction is assigned a version epoch on creation; the transaction sees modifications from older transactions and not from newer transactions. This can be implemented by having each transaction implement updates by copying the data to be updated; newer transactions use the new copy, or new snapshot, and old transactions keep using the old snapshot until they commit or abort, at which time the old snapshot is free to be deleted. The demands of scientific data systems preclude deleting older snapshots; however, the technique is otherwise very similar.

Snapshot isolation is not unique to database systems. It is known to operating system developers as Copy-on-Write [5], and algorithm designers as Read-Copy-Update [11]. There are various subtle differences between how these techniques are used in practice, particularly in how old versions are removed when they are no longer needed. The basic concept, however, is the same for all three.

The success that Git has had in the software-development arena lends credence to its idea that delta chains should not necessarily correspond to version chains. We will consider alternative algorithms for developing efficient delta chains based on Git's model. Because this is a central database system, rather than a decentralized version-control system like Git, we will model the snapshot-isolation implementation in this system after that in PostgreSQL, rather than that in Git.

Chapter 3

Programming Interface Design

Over the course of this research, we study a variety of different compression and data-manipulation techniques, and test their performance on a variety of data sets. This testing will be performed by writing a series of sample workloads that operate against a common API. It is therefore necessary to design an API that supports all necessary operations against stored matrices, and that allows for efficient implementation of all of the various techniques that will be studied.

The proposed API must provide the ability to add new versions of a matrix, and select data from past versions of a matrix. It does not need to support updating versions in place, since past versions are immutable; nor, for the same reason, does it need to support deleting. Hence, there are two basic classes of operations, Insertions and Selects.

This implementation makes the assumption that users will submit full updated versions of their matrices at a time, rather than storing a mutable head and modifying it until it is ready to be committed as a version. The SciDB database system already provides facilities for working with large mutable matrices in this way.

3.1 Definition of a "Matrix"

Thus far, a "matrix" has been specified only as an object that stores grid-like dense or sparse data. This API specifies the data type of a matrix as follows:

The *Matrix* data type stores an N-dimensional 0-indexed array of values of a single fixed existing data type. The initial implementation supports matrices using 8-, 16-, 32-, and 64-bit signed and unsigned integers. A matrix can be specified as either "sparse" or "dense". A dense matrix is stored, in memory and on disk, in row-major (or "C") form: Each row of values is stored in order, so that one can index into the array with arithmetic as follows: For an N-dimensional array with a shape vector S of length N indicating its size in each dimension, consider an index vector V, also of length N. If $N = 1$, the matrix cell to be read is $V[0]$. If $N = 2$, the cell to be read is $V[0] + S[0] \cdot V[1]$. If $N = 3$, the cell to be read is $V[0] + S[0] \cdot (V[1] + S[1] \cdot V[2])$, and so on.

To illustrate, consider a 2-dimensional 5×4 matrix:

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 \end{bmatrix}$$

In row-major form, this matrix is linearized as follows:

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 \end{bmatrix}$$

For a 5×4 array, $S = [5, 4]$. Say we want to go to index $V = [0, 2]$. The 5×4 array will be stored as 20 consecutive values; one row of 5 values, followed by a second row of 5 values, followed by a third, followed by a fourth. To get to row 2, we must therefore seek two rows into the array, or to element $S[0] \cdot V[1] = 5 \cdot 2 = 10$. Then, to get to the 0'th element in this row, we seek $V[0] = 0$ elements into the row, and return the corresponding value. Recall that we are working with 0-indexed arrays, so index 10 in the above array has a value of 11.

In two dimensions, there is an alternate storage ordering, column-major (or

"Fortran") ordering, that stores columns serially rather than rows. Indexing is then achieved by reversing the orders of V and S in the above algorithm, or following them from end to beginning rather than beginning to end. So, for example, the above matrix would be serialized as follows:

$$\begin{bmatrix} 1 & 6 & 11 & 16 & 2 & 7 & 12 & 17 & 3 & 8 & 13 & 18 & 4 & 9 & 14 & 19 & 5 & 10 & 15 & 20 \end{bmatrix}$$

In this case, we would seek to index $S[1] \cdot V[0] + V[1] = 0 + 2 = 2$ to get the element at $[0, 2]$; index 2 above is in fact equal to 11.

In higher dimensions, any axis could be serialized first, second, third, etc. Future versions of this library may offer these alternate serialization orders as options.

Sparse matrices are stored in coordinate-vector form, as a collection of (coordinate vector, value) pairs, sorted by coordinate (by the value of the first coordinate index, then the second, etc). For example, the matrix

$$\begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 1 & & & \\ & & 2 & \\ & 8 & & \\ & & & 9 \end{bmatrix} \end{array}$$

might be stored as

$$\begin{bmatrix} 0 & 0 & 1 \\ 1 & 2 & 8 \\ 2 & 1 & 2 \\ 3 & 3 & 9 \end{bmatrix}$$

Coordinates with no corresponding pair (and, therefore, no corresponding value) will be assigned the value 0 for matrices of integer data type. The pairs will be writ-

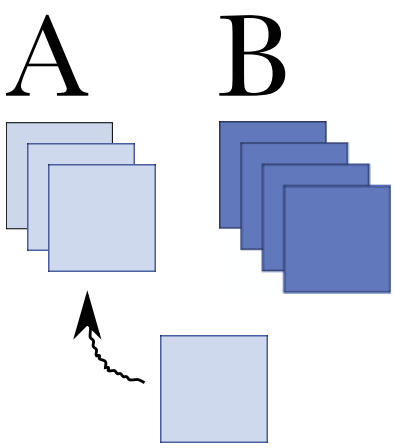
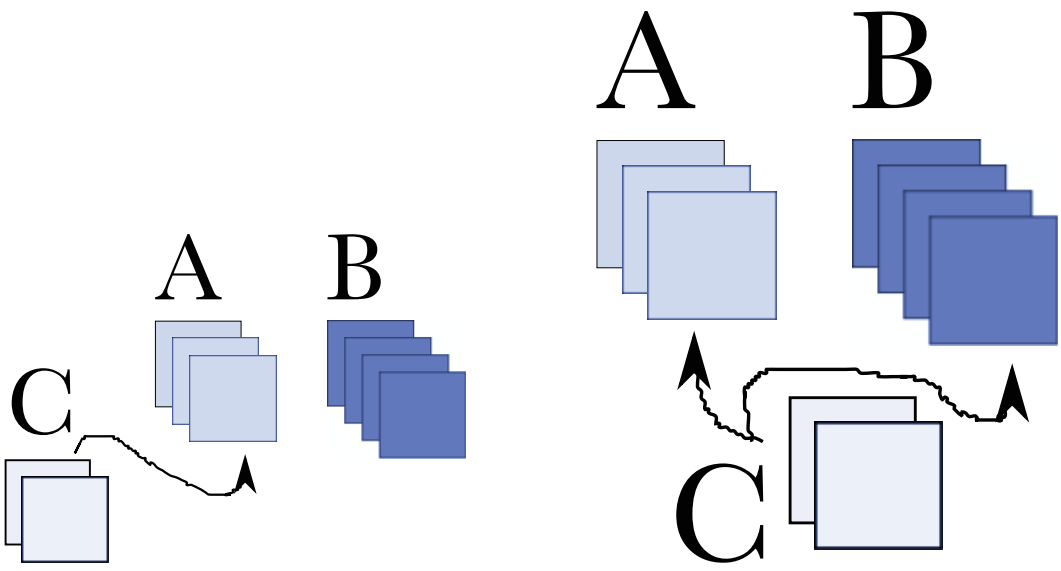


Figure 3-1: Matrix Append

ten out to disk serially. Indexing into a sparse matrix requires binary-searching through the coordinates to find the index in question. Because this may be expensive, in-memory sparse arrays may have an auxiliary hash-map structure to allow $O(1)$ indexing. This structure will only be generated if explicitly requested, and is not needed for operations like adding two arrays where it is efficient to scan the coordinate vectors sequentially.

This is just one of a wide variety of possible ways for storing sparse matrices [15][4]. Similarly to row- versus column-major form, the above pairs can be sorted by looking at the coordinate indices in any order. More broadly, values could be stored in a dense array which is then run-length encoded to eliminate wasted space due to unpopulated cells; data can be stored using a multidimensional linked list to allow for rapid insertion; diagonal matrices can be stored as a single vector representing the values along the diagonal; or any of more than a dozen known common formats. Each variant has a type of problem that it solves particularly efficiently. However, coordinate form is simple and has been shown to be relatively efficient for many types of operations [15], so it will be the default data format for this library.



(a) The last array in series A is branched to form a new series C

(b) The last array in series A and B are merged to form a new series C

Figure 3-2: Matrix Branching and Merging

3.2 Matrix Insertion

The *Insert* operation takes a complete dense or sparse matrix, and adds it to the database, as illustrated in Figure 3-1. In its typical form, it appends the matrix to a specified series. If the specified series does not exist, the Insert operation creates the series, and the matrix is added as the first element. The Insert operation also takes a "matrix name" as an argument, to indicate the matrix this version belongs to, and it returns a "Version ID" object that uniquely identifies the newly inserted matrix. If no matrix exists with the specified name, the new matrix object is inserted as the first version for that matrix.

The versioning system supports a *Branch* operation that is related to Insert, except that it takes an explicit parent matrix as an argument. Branch operates identically to Insert except that, as shown in Figure 3-2, the database system annotates the matrix as being derived from the specified parent, rather than from its predecessor within a series. The Branch operation is typically used to create a new series. It is intended for use-cases where a user has, for example, one raw-data matrix, but

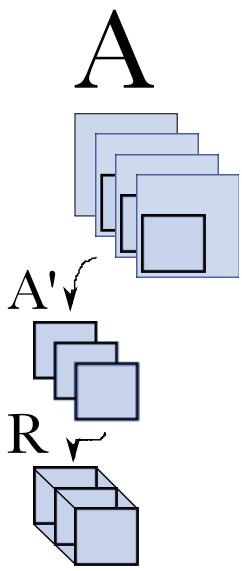


Figure 3-3: Matrix Sub-Select
A region is selected from the most recent three 2-dimensional arrays in series A. The region is selected from each array to form a temporary series A'; then combined to form a three-dimensional array R, which is ultimately returned.

wants to "branch" off from that raw-data matrix with a variety of analyses.

3.3 Matrix Select

The *Select* operation takes on four forms. In its first form, it takes a matrix name and a version ID, and returns the specified matrix object.

In its second form, it takes a matrix name, a version ID, and two coordinates in the matrix representing two opposite corners of a hyper-rectangle within the specified matrix. One corner is the corner closest to the origin; the other is the corner farthest from the origin. It returns the sub-matrix contained within the specified hyper-rectangle.

In its third form, it takes a matrix name and an ordered list of version IDs. Given that the specified matrices are N-dimensional, it returns an $N+1$-dimensional matrix that is effectively a stack of the specified versions. So, for example, in MATLAB notation, if matrix A were returned, $A(1,:)$ would be the first version selected;

$A(2,:)$ would be the second version selected, etc.

Its fourth form is a combination of the second and third forms: It takes an ordered list of version IDs and two coordinates specifying a hyper-rectangle on top of a version. It queries the specified ranges from each of the specified versions, then stacks the resulting matrices into a single $N+1$-dimensional matrix and returns it. This form is illustrated in Figure 3-3.

3.4 Querying Schema and Metadata

The versioning system supports a *List* operation, that returns the names of each series currently stored in the system. It also supports a *Get Versions* operation, that takes a series name as an argument, and returns an ordered list of all versions in that series.

There are various accessor methods to determine properties of a matrix, such as its size and whether it is a dense or sparse matrix, without reading the whole matrix from disk. They take a series name and a version ID as arguments. Finally, our system include methods to return all of the parentage information set by the Branch command. These also take a series name and a version ID as arguments.

3.5 Transactions

The proposed API supports transactional operations in a limited form.

A large fraction of the data stored in a versioned database is immutable. Because it cannot be updated or deleted, transactions are not needed to guarantee read or write safety.

When inserting new versions, "Begin", "Commit", and "Abort" operations are available, and must be used. These operations provide the traditional transactional guarantees that no other active queries can see the results of operations that occur during the transaction until the transaction has successfully committed, and the current query cannot access newly-inserted versions until it commits or aborts.

This allows multiple atomic inserts, and multiple atomic reads of the list of available versions.

In the current system, if two transactions attempt to insert versions to the same array at once, one of the two transactions will abort with an exception. This is admittedly a very simple mechanism. This code's focus is on compression, rather than concurrency; its benchmarks do not generally test concurrent array inserts. Future implementations are encouraged to allow for a wider array of options, such as locking arrays at transaction-creation time so that clients can choose to wait until an array can be safely edited, or for simultaneous inserts to result in branches, or other ideas depending on the demands of the users of the system.

3.6 Chunking and Colocation

Chunking is the ability to break a large array into multiple files on disk. Colocation is the ability to combine multiple small arrays into a single file on disk. This library does not provide runtime configuration for either ability, so neither feature has a corresponding API; but either feature can be enabled when the library is loaded. Both are discussed in more detail in Section 4.4.

3.7 Basic Structure

The basic structure of the reference library that implements this functionality is shown in Figure 3-4. The library works as follows: API calls are executed against the versioned storage engine. If chunking is enabled, API calls are passed into a thin chunking layer that splits input commands up among multiple target matrices and keeps track of a mapping between large matrix objects as seen by the outside world and the small chunks that compose them, as seen by the versioned storage engine.

In order to execute a typical Select statement, it will read the location and status of the corresponding matrix from the metadata store. If the object is collocated with

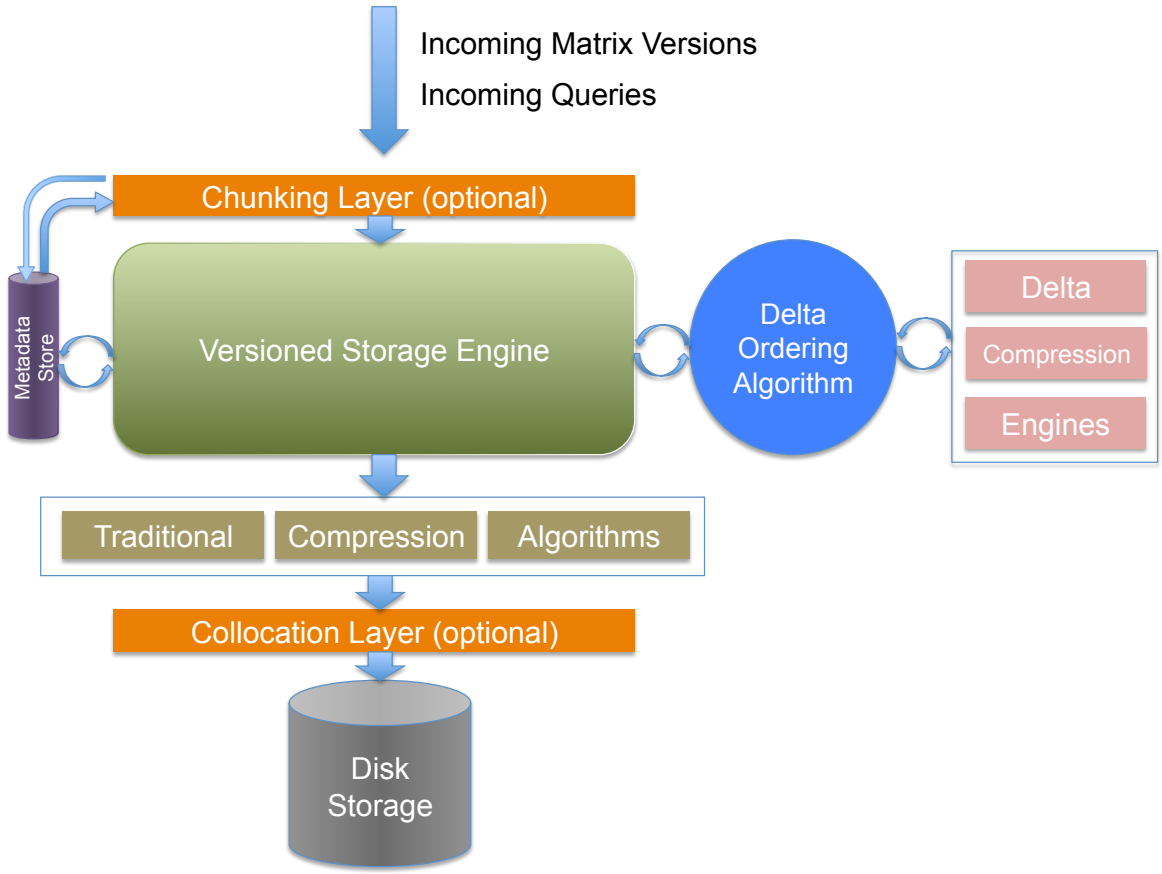


Figure 3-4: Structural Layout of the Compression Library

several other objects, the appropriate fraction of the storage file will be selected. If the matrix is stored in compressed form, it will be uncompressed by the appropriate algorithm. If the matrix is differenced against another matrix, the appropriate delta compression engine will be used to reassemble it, possibly requesting other matrices from disk in the process.

In order to execute a typical Insert statement, the new matrix is passed into the chunking layer if it is enabled, or directly to the versioned storage engine if not. The versioned storage engine will pass it straight to disk, optionally compressing (but not differencing) it along the way. The delta-ordering algorithm will periodically run and check for new arrays, and decide if or how to difference them against existing arrays. This allows differencing algorithms to take into account all existing matrices when performing differencing and deciding which matrices should optimally be differenced against each other, without requiring expensive calls into the metadata store every time a new matrix is inserted.

The versioned storage engine exposes an API in the Python programming language that any alternative engine can implement. In fact, the chunking layer is simply another storage engine implementing the same API, that happens to call into the versioned storage engine for most of its operations. Two other alternative storage engines have also been developed, using the Git and Subversion version-control systems to handle backend storage and compression.

Chapter 4

Differencing Methodology and Options

The library that implements this API provides several different types of data storage and compression. Specifically, it provides several different types of delta compression and propose two matrix-specific delta algorithms. It also provides traditional single-matrix compression algorithms, periodic materialization of delta chains, and some variants of the basic disk-storage model for arrays.

For delta-generation algorithms, BSDiff is one option, as well as three new methods: matrix bit-reduction, sparse deltas, and hybrid storage (with both traditional and MPEG-based differencing). For traditional compression algorithms, run-length encoding, duplicate elimination, and an encoding based on the Lempel-Ziv transforms [25] are considered.

4.1 Generic Delta-Compression Algorithms

A variety of generic binary differencing algorithms currently exist. The *BSDiff* algorithm has been shown to work well on a variety of different inputs without being so specialized that it cannot operate on arbitrary data [13], so it will be tested here.

BSDiff is a relatively expensive algorithm on large files that have substantial

differences between them. Its author did not analyze its runtime, but it is based on an algorithm that runs on $O(ND)$ time [12], where N is the sum of the sizes of the two input files and D is the size of the output file, and so is likely at least that expensive. In some situations, a faster algorithm could be preferred.

4.2 Matrix Delta-Compression Algorithms

Differencing integer matrices can be done in $O(N)$ time via cell-wise subtraction of the two matrices.

4.2.1 Matrix Bit-Reduction

Once we have subtracted matrices, there is some question about how to most efficiently store the difference matrix. One approach, the *matrix bit-reduction* approach, is to guess that the maximum absolute value of any single cell will have decreased due to the subtraction. We can scan all values in the array to determine the maximum value, to verify that this is indeed the case, and calculate the number of bits needed to store this new maximum value. If, for example, two 32-bit-integer matrices were subtracted yielding a maximum difference value of 200, the difference could be stored in an 8-bit-integer matrix, cutting its size by a factor of four. So, the size difference of the reduced delta can be calculated by

$$\text{number of total values} \cdot (\text{original size(value)} - \text{new size(value)})$$

4.2.2 Sparse-Delta storage

Another approach is to assume that most values in the difference matrix are zero. If this is the case, the difference matrix can efficiently be stored as a sparse matrix where all unspecified values are assumed to be zero. For differences of sparse matrices, this will happen automatically if sparse subtraction is implemented to not write out zero values. For dense matrices, this requires converting the dense

matrix to a sparse matrix, the *sparse-delta* approach. This will be efficient iff sufficiently many cells are zero that the space savings due to eliminating them offsets the space cost of storing a coordinate along with each value. This will be the case if the size difference between the sparse delta and the original delta is less than zero, or if

$$\text{sizeof}(\text{value}) \cdot \#\text{ total values} - (\text{sizeof}(\text{coord vector}) + \text{sizeof}(\text{value})) \cdot \#\text{ nonzero values} < 0$$

For a 32-bit matrix with 32-bit indices, for example, fewer than $\frac{1}{3}$ of the values in a dense difference matrix can be zero.

Dense matrices often represent raw or simulated data from sensor grids. For example, they may represent the image from the sensors in a digital camera. Adjacent sensor-grid matrices may see large differences in parts of the grid that have changed. However, they are also susceptible to noise, and so may see small changes at most cells in the difference matrix. As a result, neither of the above methods tend to perform well. As an alternative, I propose the following *hybrid storage* approach for compressing a dense difference matrix:

4.2.3 Hybrid Storage

The approach creates a counter vector, with one element for each possible size of a value in the delta matrix. In this system, with 32-bit input matrices, a counter vector would be of length three: One cell for 32 bits, one for 16, and one for 8. The approach iterates through each value in the difference matrix; and for each value, increment the cell in the counter vector that corresponds to the minimum number of bits needed to store that value. Example code for this can be found in Figure 4-1. Based on the counts in the count vector, the approach picks a threshold value. It splits the difference matrix into two matrices along that threshold: Cells with values greater than the threshold go into one matrix, and cells with values less than or equal than the threshold go into the other matrix. The matrix with

```
unsigned int gt_0b = 0, gt_8b = 0, gt_16b = 0, gt_32b = 0,
             gt_8bu = 0, gt_16bu = 0, gt_32bu = 0;
%(type)s val, abs_val;
for (blitz::Array<%(type)s, %(dims)s>::iterator i = diff.begin();
                                                i != diff.end();
                                                i++) {
    val = (*i);
    abs_val = (val >= 0 ? val : -val);
    if (val != 0) gt_0b++;
    if (abs_val > 127) gt_8b++;
    if (val > 255) gt_8bu++;
    if (abs_val > 32767) gt_16b++;
    if (val > 65535) gt_16bu++;
    if (val > 2147483647) gt_32b++;
    if (val > 4294967295) gt_32bu++;
    if (val < 0) {
        gt_8bu++;
        gt_16bu++;
        gt_32bu++;
    }
}
```

Figure 4-1: Simple Matrix Bit-Depth Counter
Written in C++ with Python templates; compiled using Python's scipy.weave library [20]. Calculates the values for the counter vector.

large values will receive fewer elements, and will be compressed with the sparse-difference approach above. The matrix with small values will tend to need fewer bits per value, and so will be compressed with the bit-reduction approach above.

To re-combine these two matrices, we can simply add them together cell-wise.

To determine the threshold value, we need a way to measure the size of the two output matrices. As noted above, the delta matrix size can be calculated by the expressions

$$\text{number of total values} \cdot (\text{original size}(\text{value}) - \text{new size}(\text{value}))$$

and

$$\text{sizeof}(\text{value}) \cdot \text{\# total values} - (\text{sizeof}(\text{coord vector}) + \text{sizeof}(\text{value})) \cdot \text{\# nonzero values} < 0$$

In this case, the number of zero values in the sparse matrix is equal to the number of values that would get assigned to the dense matrix as indicated by the counter vector, and the new size of a value in the dense matrix is equal to the number of bits needed to store the chosen threshold value. So, it's possible to calculate the total expected disk usage for any given threshold size given only the counter vector and the equations above. To calculate the optimal threshold, consider each possible value size in the counting vector as a threshold; calculate the data sizes resulting from each possible threshold value, and pick the threshold that yields the smallest data size.

There are a maximum of four possible thresholds if using the matrix sizes in this system, or 64 if values are being truncated to the nearest bit; so in terms of runtime, this processing stage takes $O(1)$ time and all that remains are two linear-time scans of the matrices, leading to an $O(N)$ runtime.

4.3 MPEG-Based Differencing Optimizations

The contents of two consecutive matrices may be very similar but offset slightly, as was found in motion video and adjusted for by the MPEG-2 video standard [9]. I therefore propose a means of compensating for potential motion from one version to another based on the technique used in the MPEG standard: Say that matrix A is being differenced by matrix B. Divide A into regular square (or cubic, or N-hypercubic for higher-dimensionality matrices) chunks of edge size K, for some K that will be empirically determined. For each chunk in A, difference it against the corresponding region in B, and against all equally-sized square regions within a radius K of the original block's location. Essentially, slide the square around until the best fit is found. The "best fit" is defined as the offset where the maximum value in the difference matrix between the chunk and the corresponding region in B is minimized. Once this is found, we store the difference between the chunk and the minimizing offset region in B. Additionally, in an auxiliary data structure, we store the offset that was needed for this chunk, so that A can later be reconstructed by adding each chunk to the appropriate region in B.

This algorithm results in $O(K^2)$ differencing operations on each of the $O(N)$ chunks. As such, it has the potential to be an expensive operation if K is large. However, if K is small, large motions will not be captured.

4.4 Chunking and Colocation

"Chunking" is the process whereby a single large array is split up into multiple files, each representing a region of the larger array. Particularly when using data compression, such that it is not possible to quickly seek into arbitrary locations in an array stored on disk, chunking can substantially decrease the amount of data that must be read from disk if only a portion of an array is needed.

"Colocation" is the process of storing multiple arrays consecutively on disk, often by placing them in the same file. If arrays that are frequently accessed simul-

taneously are collocated, this reduces the number of disk seeks required to retrieve any given array. When arrays are small, the constant-time cost of a disk seek per array can dominate the linear-time cost needed to read the array, so cutting down on seek costs can be an important optimization.

When storing a versioned matrix, chunking and colocation can be seen as two sides of the same coin: The versioned N-dimensional matrix, with many versions in its history, can be thought of as a single $N + 1$-dimensional matrix where all of the versions lie along one axis. Colocation of versions then simply becomes a different type of chunking.

However, with delta compression, this theory does not clearly represent the practical implementation of a storage system: Delta files are blobs that may not be readily split into chunks. Even if a delta is split into chunks, it is possible that each individual chunk could have been differenced more efficiently as a chunk than along with the whole array. Consider the case where one region of a versioned matrix frequently varies a great deal, and another region sees only small occasional variations. The delta chunks from the low-variation region could likely be stored using fewer bits than the delta for the whole array would need, for any given delta between two versions. Therefore, when working with delta compression, it is preferred to perform chunking, then differencing, then colocation.

Chunking and colocation each impose some amount of overhead, both in terms of tracking the number and arrangement of chunks and reassembling chunks as needed. Both chunking and colocation also add a number of complicating factors, such as increased data-load time and increased risk of file fragmentation in some environments, that pose challenges for careful benchmarking.

4.5 Delta Ordering and Performance

There are many ways to produce a delta tree from a given set of arrays. Given N arrays, there are $N!$ simple delta chains, and an even greater number of delta trees, that can be produced simply by ordering the deltas differently.

The versioning system provides two natural orderings: From the oldest to the newest version, and from the newest to the oldest version, with materializations as needed to account for branches in the system. There is some reason to believe that this ordering will be a good ordering for efficient deltas. If one assumes that a small number of changes will be made between adjacent versions, then the corresponding deltas will likely also be small. However, there is no guarantee that this is the optimal ordering. For example, matrix versions from a periodic data source might want matrix objects that are one period apart to be differenced against each other.

A brute-force search for the optimal delta ordering would run in exponential time, because there are exponentially many ordering possibilities to consider. Such a search would be computationally infeasible for systems containing large numbers of arrays.

Work is being done presently on an algorithm that is more efficient, at least in typical cases [17]. These algorithms are still being analyzed. However, there are some important questions that such an algorithm would need to consider, and a number of heuristics that work quite well in common cases.

The most important question to consider is "how is 'optimal' defined?" In keeping with an emphasis on performance and disk usage, one could define it to mean that a data set is stored using as little space as possible, or that a data set is stored such that a given query plan executes as quickly as possible against the data set.

4.5.1 Space-Optimal Algorithm

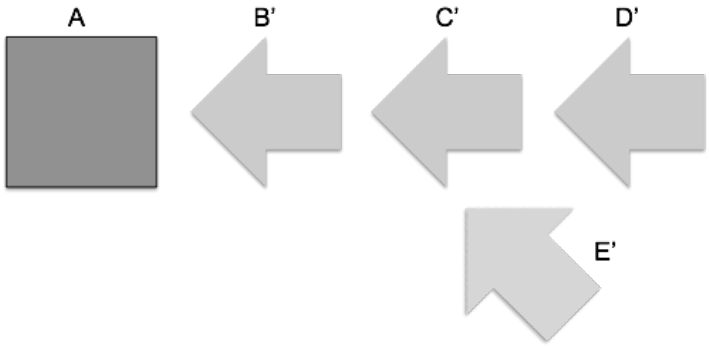
The Git version-control system has a simple heuristic that optimizes for disk usage. Stored objects are sorted by a number of similarity metrics. Then, each object is differenced against its N nearest neighbors (for a configurable value of N), and the smallest differences are stored.

Unfortunately, Git's similarity metrics include such items as file size and file type, which will often be identical for this versioning system. It therefore becomes necessary to find a similarity metric, that can be used to determine how well two arrays will difference against each other.

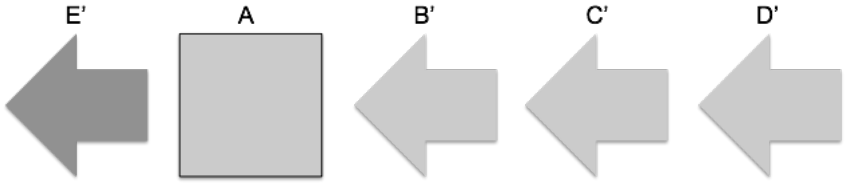
The simplest approach is to simply try all N^2 possible differences, and calculate their sizes. This operation requires $O(N^2K)$ time, where N is the number of versioned arrays and K is the expected size of an array. In essence, it requires N sequential scans of the entire database. For a very large database with many arrays, this is unlikely to be feasible. For example, a typical modern hard disk might read data at approximately 50MB/sec. Say that 1,000 1GB arrays were stored on a 1TB hard disk. It would take approximately 11 hours to read the full contents of the disk. To read it 1,000 times would require roughly 462 days.

A cheaper heuristic might be to create a fingerprint of some sort that identifies an array, such that two arrays with similar fingerprints will tend to be similar. There are many possible fingerprints; the maximum and minimum values in the array, the average and standard deviation of the array, a reduction of the array into a small constant number of bytes using sampling or image-processing algorithms, or others.

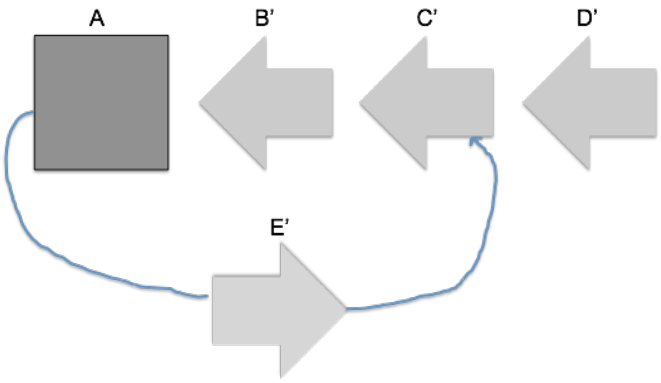
Another approach to differencing is a greedy algorithm: Arrays are inserted one-by-one into a difference tree; on each insertion, the inserted array is differenced against each existing array and vice versa, and the differencing that saves the most space is selected and saved. This algorithm is suboptimal because it doesn't consider the possibility of simultaneously differencing an existing array against the new array and the new array against an existing array. This possibility is not permissible without modification because it creates a cycle in the tree. See Figure 4-2 for an illustration of this effect. However, it is possible that the two new differences are both more beneficial than some set of existing differences in the current tree; and that it would be possible to select a pair of new differences that forms a cycle containing a more-expensive difference that could be eliminated to break the



(a) When adding array E′, it can be differenced against any existing array.



(b) Additionally, any array, including the materialized array A, can be differenced against it.



(c) Doing both, however, will create a loop that must be broken by materializing another array.

Figure 4-2: Matrix Delta, Delta Chain

cycle. Resolving this edge case correctly can be computationally expensive. Hence, this heuristic is proposed as a less-expensive approach to finding a good, though not necessarily optimal, arrangement of arrays.

4.5.2 Performance-Optimal Algorithm

For optimal performance, the simplest case is to note that deltas increase query time, because they require reading an older array and a difference rather than simply reading the current array. So, one could assert that having no deltas is performance-optimal. This is correct for a simple differencing system that does not support range selects. However, if a user can select 10 arrays at once in a single query, it may well be faster to have 9 of those arrays differenced against each other or against the tenth array, because that way, less data need be read from disk. Similarly, if the query system has a buffer pool that fits a subset of the data, a query might only need to read a delta from disk and a commonly-accessed full array from the buffer pool.

In these cases, it can be optimal to store an array more than once, as both a difference against one or more other arrays and as a fully-materialized array. To store the globally optimal set of arrays, one could consider all possible queries against the database and store the space-minimizing set of data for each such query so that it can be read in as quickly as possible.

This method does not interact well with a buffer pool, however, as there will be many more arrays on disk so the probability of a needed array already being in memory becomes smaller. This could be mitigated to a significant extent with a query planner that considers the current state of the buffer pool: If more than one delta chain can reconstruct the requested array, only consider the disk cost of reading arrays that are not in the buffer pool. This cost varies based on whether the buffer pool only stores fully-materialized arrays or whether it stores deltas; if it stores deltas, arrays in the buffer pool will still have some associated cost due to copying and reassembling the deltas in memory, but if it stores fully-materialized

arrays, fewer arrays will fit in the buffer pool.

Additionally, this approach can potentially use very large amounts of hard disk space, if a large number of redundant deltas are stored. A simple solution to this problem is to not store redundant deltas. In this case, without a buffer pool, the performance-optimal solution will tend to be the set of materializations that minimizes the total amount of data read from disk for the given set of queries, assuming that CPU cost is small.

4.5.3 Performance-Optimal with Disk Constraint

A user may still have disk-space constraints that would preclude materializing arbitrarily many arrays. This leads to the compound optimal condition of a performance-optimal solution given a constraint on the amount of disk space used. The only solution we know to this problem is testing all possible materialization patterns against the anticipated workload. If redundant differences are allowed and the query workload contains a large number of different queries, this yields an algorithm that stacks a series of polynomial- and exponential-runtime components together to produce an algorithm that is entirely unusable for most data sets.

A simplified heuristic approach to this algorithm starts with a space-optimal set of differences and a query workload. It uses a cost metric that is as follows:

$$\text{Cost} = (\text{Array Access Frequency}) \times (\text{Materialized Array Size} - \sum \text{Size of delta chain})$$

This metric tells us how much less data we would read from disk if this array were materialized, under the given query workload. Given this metric, we apply a greedy algorithm: Keep materializing the array with the largest metric score until there is no longer sufficient available disk space to do so. This heuristic is proposed as an approximate solution to finding an optimal-performance delta arrangement.

This algorithm will be suboptimal in the case where a small delta cannot be materialized and the algorithm stops as a result, but a larger delta (which would

require less space to be materialized) could still have been materialized. In general, this type of problem is quite similar to the class of packing problems, and isomorphic to the "backpack" or "knapsack" problem in particular [8]. The backpack problem can be phrased as "given a backpack that can carry up to a given weight, and a set of objects with known weights whose sum exceeds the capacity of the backpack, how heavy can the backpack be without being overfull?" In this case, the "backpack" is the hard disk, and the "objects" are delta materializations, with known disk-space costs instead of known weights. All packing problems, including the knapsack problem, are known to be NP-hard [8]; the greedy approach is one simple and cheap heuristic often used to approximate the optimal solution.

4.6 Traditional Compression Algorithms

In this work, we make use of SciDB's matrix-compression library for a variety of compression algorithms, including a variety of Lempel-Ziv encoding, Run-Length Encoding, Background Subtraction, Adaptive Huffman Encoding, and Bitmap Encoding, as well the GraphicsMagick library [6] for the JPEG 2000 and PNG image-compression algorithms.

Lempel-Ziv encoding has several variants, discussed in several papers by Lempel and Ziv published between 1977 and 1978 [24][25]. However, the basic premise of all of their algorithms is similar: Look for and identify repeated patterns of increasing length in a string of data, in a dictionary of patterns that have been seen so far. If a repeated pattern is found, read the symbol (or byte, word, or other unit of data) following the end of the pattern, and add a new pattern (<pointer to old pattern>, <new symbol>) to the dictionary of known patterns. So, now there is a new pattern in the dictionary of known patterns to match against, that is longer than the previous pattern.

Run-Length Encoding is a simple encoding mechanism where duplicate values are replaced with tuples of the form (value, number of repetitions). If a value is repeated many times, this sort of compression can be highly efficient.

Background Subtraction is a similar mechanism to Run-Length Encoding, except that its goal is to eliminate null values: It stores a buffer representing a sequence of values of arbitrary length (prefixed by the length of the buffer), followed by an index indicating how many zero values the buffer was followed by.

Adaptive Huffman Encoding, also known as Dynamic Huffman Coding [21], is a variation of Huffman codes where the probability distribution of the symbols in the encoded alphabet need not be known in advance. A Huffman code is a binary tree that encodes an alphabet, or a collection of possible symbols or values (such as all of the 256 possible values of a byte). One symbol is stored in each leaf of the tree. The tree is deliberately unbalanced; symbols that appear more frequently are stored in leaves closer to the top of the tree. Note that a given leaf of a tree can be uniquely identified by a binary string that represents a descent of the tree, where '0' indicates "take the left child" and '1' indicates "take the right child". So "001" would, for example, represent the root's left child's left child's right child. With this encoding, a string of symbols can be encoded as a string of bits representing indices into the Huffman tree. Adaptive Huffman Encoding dynamically reorganizes its tree in a deterministic way based on the symbols that it has seen so far.

Bitmap Encoding is a simpler cousin to Adaptive Huffman Encoding: Given a dictionary of N symbols, each symbol gets assigned a number $0 \ldots N-1$. This number can be stored in $\lceil \log N - 1 \rceil$ bits. Therefore, the input string can be described as the corresponding string of $\lceil \log N - 1 \rceil$-bit words, plus the dictionary mapping numbers to symbols. This is particularly effective where a field may use a large amount of storage space (such as a string) but where values for this field may only be drawn from a very small pool of possible values.

The JPEG 2000 standard [18] is a blanket standard that covers a variety of means of compressing images. Some of these methods are lossy; others are lossless. All of them are based on the ideas of breaking images into tiles, and using wavelet transforms recursively to break images down into more and more smaller and smaller sub-images.

The PNG image-compression algorithm [1] is a layered algorithm, in that com-

pression occurs in multiple stages. First, a filtering stage occurs, that is not intended to decrease the image size but is instead intended to introduce redundancy in the image data. PNG allows for a variety of filter stages, such as subtracting each pixel from its adjacent pixel in one direction. The second stage is to encode the resulting filtered image with a Lempel-Ziv compression algorithm variant.

Duplicate elimination is not strictly a traditional compression algorithm, in that, if it has any effect at all, that effect is to completely eliminate the need to store a given matrix. Quite simply, if two matrices are found to be identical, one is removed from disk and replaced with a pointer to the other. Because past versions are immutable, it is safe to eliminate duplicates from among them without taking additional steps to prevent updates from one file from propagating to another file. The MD5 hashing algorithm [14] is used to allow a duplication-elimination sweep to run in $O(N)$ time, where N is the number of stored matrices; each matrix is hashed, and only matrices with identical hashes are considered as possible duplicates.

4.7 Alternative Compression Approaches

All traditional compression algorithms operate on a linear buffer of information. Therefore, matrices of any size must be linearized before they can be compressed. Traditionally, matrices are linearized in row-major or column-major order. However, the linearization ordering need not itself be linear at all. For example, Hilbert curves [22], a type of space-filling curve that tends to cause cells that are near each other in N-dimensional space to be near each other in the resulting linearization, have been considered [23] as a means of compressing matrices. The hypothesis is that values near each other, even if they are not in the same plane or adjacent in a row-major or column-major ordering, will tend to be similar; so a linearization order that causes them to appear close to each other will tend create patterns and yield better compression.

Chapter 5

Metrics and Models

In order to test the efficacy of the proposed algorithms, it is necessary to define what it means for an algorithm to be "good"; or, more generally, to be able to rank and compare algorithms. It's also useful to be able to predict the performance of a given algorithm, to help identify and understand unexpected bottlenecks when benchmark results do not match real-world results. To do this, it's necessary to understand the constraints on a typical database system, and the properties that most affect how well a given piece of software meets those constraints.

5.1 General Constraints

At a high level, given a reliable system, users of a database care about two main things: How quickly their queries execute, and how much of an investment (in terms of hardware, engineering expertise, and anything else) they need to make in order to get the queries to run. These two factors may not be independent; often, it is possible to purchase more-expensive hardware or hire staff to tune a database system in order to speed up query execution.

Scalability is one common contemporary metric of how "good" a database system is. Scalability is important according to the above criteria because it guarantees that queries will continue to execute quickly as the data size grows. All compression techniques that have been proposed are very amenable to scalability by using

chunking to break up a large matrix and having compression operate in parallel on individual chunks. Therefore, scalability at the individual-matrix level will not be considered.

"Investment" can be broken up into two primary categories: Human resources and technical resources. Human resources are needed to manage large, complex software systems that do not self-tune. All algorithms discussed here are in some sense interchangeable; no one proposed algorithm requires appreciably more hand-tuning than any other. As a result, human resources are not affected by algorithm choice. Technical resources, on the other hand, can be affected. For example, if an algorithm requires a large amount of memory, a system housing that much memory would need to be purchased. Because scalability by chunking is being assumed, only resource bottlenecks within a single computer will be taken into account in these benchmarks.

5.2 Performance Factors

A number of factors contribute to how quickly queries execute on a given computer. In order to execute a query, a matrix (and possibly the matrix or matrices that it is differenced against) must be read from disk into memory; a series of computations must be executed to extract and possibly un-chunk the matrix, resulting in some number of copies of the matrix data in memory; and finally the matrix must be returned to the user for processing.

In this scenario, the raw data must be stored on disk, so disk space is a limiting factor. Also, the array must fit uncompressed into system memory; this is a limiting factor, but it is governed primarily by the array size and not the compression implementation, so better compression will not necessarily help a great deal. However, decompression must not use huge amounts of system memory.

The CPU serves as a bottleneck: Any CPU can execute any query, but an arbitrarily slow CPU will increase the execution time arbitrarily. Similarly, reading from the hard disk and reading from system memory will be bottlenecks.

5.2.1 Modeling Space Usage

For both in-memory and on-disk modeling, the proposed model is quite simple: Compressed data should never be larger than its own uncompressed form. For all compression code, a check will be added such that, if the compressed form would be larger than the uncompressed form, the data will be stored in uncompressed form.

In-memory operations may require considerable working space. For example, a chunked hybrid-difference array may be read in from disk in two pieces. To reconstruct the array, those two pieces must be uncompressed, creating two new small arrays; then those pieces must be added together, creating another new array; then the array that they are differenced against must be read into memory, and the difference and the source array must be added together to make a new array; then that chunk must be copied into the proper location in the larger un-chunked array. That's six total copies of the raw data. Some of these operations could be executed in-place, so it may be an overestimate; but it is at least an upper bound. To allow conservatively for additional overhead, we will assume that no implementation will ever use more than 8 times as much memory as the largest array that it stores.

5.2.2 Modeling Bottlenecks

The first step in reconstructing an array is to read all of its pieces in from disk. Reading an object from disk requires two steps. First, the disk head must seek to the location of the object. This operation requires roughly a constant time; the time is independent of the array size. Second, the disk must read the object. This takes time linear in the size of the object.

Modern hard drives require around 10ms per seek and as reading data at 50MB/s. The hard drive in the computer system used for these tests required 8.1ms per seek and read data at 60MB/s, as seen in Figure 6.1. However, assuming for the moment that these numbers are correct, note that a 10MB file will be read in 200ms.

The 10ms seek time is therefore only 5% of the read time. With arrays that are potentially many gigabytes in size, it seems that seek times should generally remain negligible. Therefore, our cost model will only take linear data-read times into account.

When executing disk operations, computer scientists like to assume that in-memory operations are much faster than on-disk operations, and so have negligible performance impact. However, on at least one preliminary test system with an old memory bus and a fast hard-disk array, memory bandwidth was found under some circumstances to be roughly three times as fast as hard-disk accesses[1]. The above model proposes that memory usage may reach as much as 6 to 8 times that of disk usage. Therefore, memory access time clearly cannot be seen as negligible.

Taking these factors into consideration, a reasonable model would seem to consider the expected time needed to read the specified data, added to the expected time spent writing to newly-allocated memory buffers. Other factors should have a negligible contribution to this total.

5.3 Test Metrics

Given our expectations from the above sections, it seems that the two most-interesting factors to consider and to try to predict when modeling algorithms are how much time they take to execute and how much disk space their stored output uses. The test code used herein is therefore instrumented to measure those two things.

Various other aspects of the algorithm and library are instrumented as well. For example, disk activity is tracked, to verify that the algorithm is reading data as expected; and the implementation has been monitored with profiling tools to verify that it is producing the expected number of array copies.

[1]To test memory bandwidth, a simple program was written that malloc()'ed a 1gb buffer, then memset() the buffer to 0. The code was run repeatedly, to force the memory to be otherwise-unallocated. In steady state, the memset() call took 1.5 seconds; 1.5 s/GB works out to roughly 670 MB/s. A subsequent call to memset() on the same buffer took 0.25 seeconds, or roughly 4GB/s, indicating that a bottleneck is probably present in the initial allocation of fresh pages. The "hdparm" hard-disk utility indicates that the connected RAID array can read at 220MB/s.

Chapter 6

Evaluation and Benchmarking

Because of the number of permutations of different delta algorithms and compressors and chunking schemes, this experimentation took a two-stage approach. First, a rough pass was taken to test for and eliminate any implementations that were demonstrably inferior to other available alternatives. Second, the remaining implementations were analyzed in greater detail.

6.1 Implementation

This library, and the test suite around it, were written primarily in the Python programming language. Performance-sensitive operations called out into existing Python modules that used compiled languages (Cython, C, or Fortran), or, in the absence of an existing module, custom C++ code was written to perform the operation, using either the Boost.Python C++/Python binding library or the scipy.weave Python module for writing inline C++ in Python.

Dense matrices were represented using the NumPy library. Sparse arrays were initially represented using the "sparse" submodule of the SciPy library. However, this module only supports two-dimensional sparse matrices. Support was heavily expanded using a custom implementation that stored coordinates in NumPy index vectors. The implementation was optimized using NumPy vector operations and built-in functions to remove all tight loops from Python code and to minimize

unnecessary copying of data in memory

Test cases were implemented using the Python unittest framework. A handful of shell script code was written to allow the batching and results-logging of tests.

Metadata in this system, such as mappings from matrix name and version number to on-disk filename and compression status, was stored in a PostgreSQL database. PostgreSQL ultimately proved to be a substantial performance bottleneck; as a result, much of the metadata was duplicated in system memory.

A few microbenchmarks were also implemented in order to test particular specific performance properties of the test system. These were written in the C programming language.

The implementation of the library contained around 5,400 lines of Python code, 150 lines of C and C++ code, and 630 lines of SQL, mostly schema definitions with comments. The test cases contained an additional 2,100 lines of Python and 380 lines of shell scripts.

Tests were executed primarily (and unless otherwise noted) on the Caneland workstation at the MIT Computer Science and Artificial Intelligence Laboratory. Caneland is a general-purpose workstation, outfitted to run a variety of different kinds of tests. It has several hard drives and a RAID array, all with varying performance characteristics. For these tests, a stock 7200RPM hard drive was used. Caneland also has 16 CPU cores, each running at 1.6GHZ. Most tests ran on a single core, though a few parallel tests were conducted across all 16 cores. Additional system specifications and microbenchmark results can be seen in Table 6.1.

6.2 Data Sets

A variety of data sets were used over the course of this testing. Some were contrived; however, three real-world data sets were selected as well.

The first data set was from the **National Oceanic and Atmospheric Admin-**

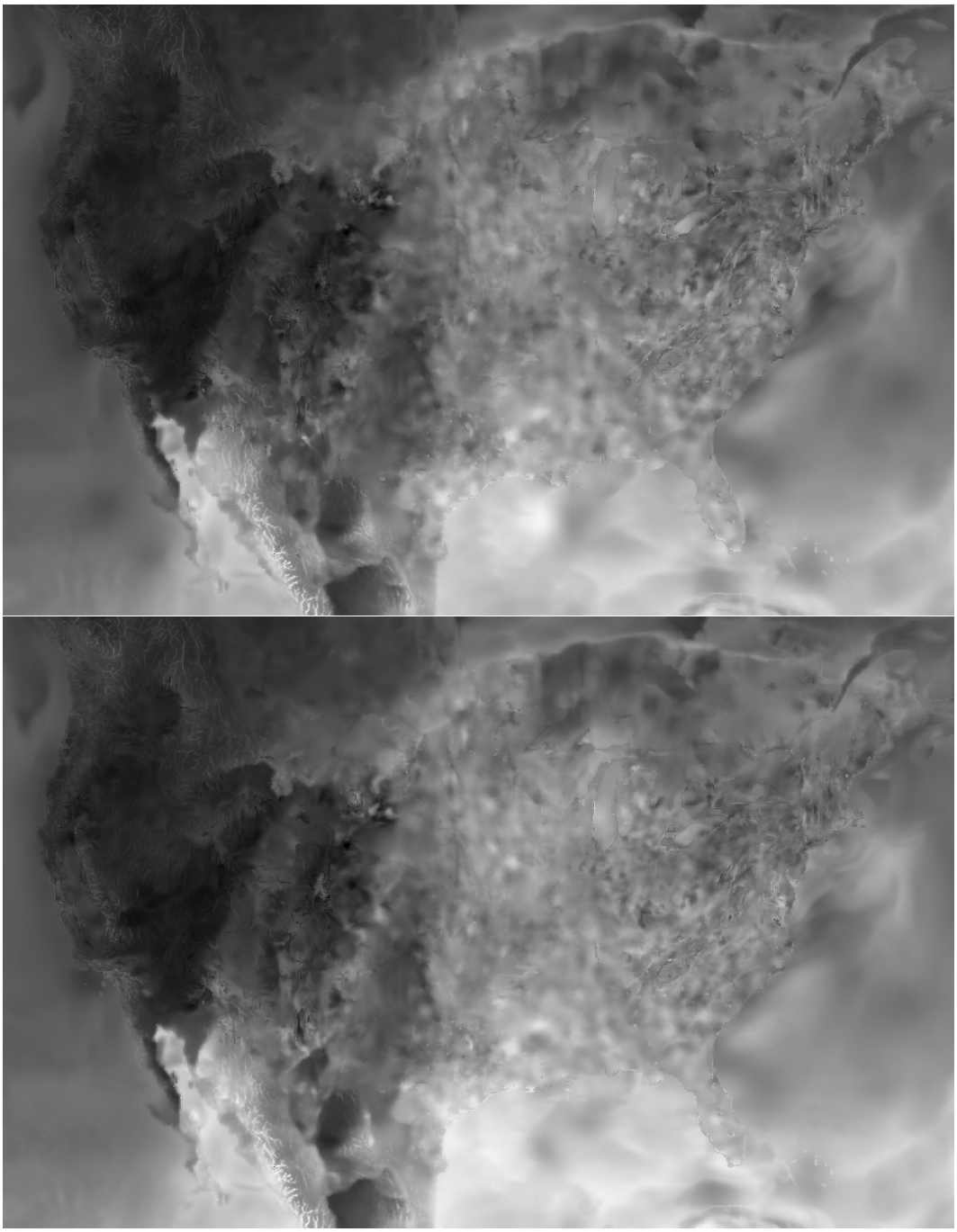


Figure 6-1: Two consecutive matrices from the NOAA weather data Measuring Specific Humidity on August 30, 2010, at midnight and 1am (respectively). Measurements are scaled to an 8-bit greyscale color space for display. Note that the images are very similar, but not quite identical. Note for example the slight change in humidity off of the coast of Long Island.

Figure 6-2: Two consecutive tiles from the OpenStreetMaps map data Showing the Kendall Square area of Cambridge, Massachusetts, including the MIT Stata Center, from December 23 and 30, 2010 (respectively). Note the change in signage along Memorial Drive.

istration sensor data for the United States. Specifically, their "RTMA" data, for August 30 and 31, 2010. This database contains sensor data measuring a variety of conditions that govern the weather, such as wind speed, surface pressure, humidity, and others, at each of a grid of locations covering the continental United States. Each type of measurement was stored as its own versioned matrix, with versions added to each matrix in chronological order. See Figure 6-1 for an example. The data typically has approximately 9 matrices per sample, with samples every 15 minutes, for a total of 1,365 matrices. For the rough-pass tests, only the first 10 time intervals were considered, for a total of 88 matrices.

The data in this data set is natively provided in floating-point form. At present, only differencing of integers is supported by the reference implementation. As a result, the data was converted to integer form by multiplying or dividing each matrix by 10 as needed to cause the maximum single cell value's magnitude in the matrix to be between 10,000 and 100,000; then truncating to a 32-bit signed integer.

The second data set was from the **Open Mind Common Sense ConceptNet** network. This network is a graph in which nodes, representing "concepts", are connected by labeled, weighted edges. The structure is stored as a large, highly sparse square matrix. Only the latest ConceptNet data is accessed regularly, but their server internally keeps snapshots for backups; the benchmark data set consisted of roughly-weekly snapshots from all of the year 2008.

ConceptNet is constantly growing, so its size is not exact; however, its representing matrix has an edge size of around 1,000,000; with on the order of 430,000 data points populated within that matrix over the course of 2008. Each data point is a 32-bit integer.

The third data set was a collection of **Open Street Maps** map tiles, representing a street map of Boston, Massachusetts and its surroundings. A map tile, such as the tiles in Figure 6-2, is a 256x256-pixel image that is a fragment of a larger image representing a map of the world. Each version consisted of a 1GB-uncompressed

map image from a region overlooking Boston, Massachusetts. Map tiles from Open Street Maps' zoom level 15 were used (one cell corresponds to roughly 4.77 meters at the equator), from GPS coordinates (-72.06, 41.76) to (-70.74, 42.70). Tiles were converted to a three-dimensional matrix where the first two dimensions represent the horizontal and vertical axes of the image and the third dimension is three units deep, one level corresponding to each of the red, green, and blue components of the image in the RGB colorspace. This approach and these GPS coordinates yield an image that is several gigabytes in size. It was trimmed to approximately 1G by selecting a $\lfloor\sqrt{\frac{10^9}{3}}\rfloor \times \lfloor\sqrt{\frac{10^9}{3}}\rfloor \times 3$ matrix centered on the large existing matrix.

16 versions of this data were stored, one per week for the last 16 weeks of 2009.

The NOAA weather data represents an array of different types of data. Much of it is very dynamic data: Almost all cell values change from measurement to measurement, so deltas will tend to be very dense. It does also have some relatively static data, where there are few changes from version to version. The ConceptNet data not only has sparse deltas; the raw data itself is sparse. The Open Street Maps data tends to have relatively sparse deltas because maps are fixed-color line drawings; also, each map array is much larger than each array in either other data set. As such, these three data sets represent a spectrum of different types of raw data.

6.3 Micro-Benchmarks

Prior to executing any tests on real data, a number of microbenchmarks were run in order to gather some more information about the properties of the test environment and tools.

One micro-benchmark ran the code in Figure 6-3, to test the performance of memory buffers on the host system. The output of the program indicated that the first call to memset() took 1.37 seconds to zero a contiguous 1GB buffer, and the second call took 0.38 seconds to re-zero the same buffer. This indicates data-

Table 6.1: Test System Performance Properties

System Architecture	x86_64 (GNU/Linux 2.6.32, Ubuntu Lucid)
System CPU	Intel E7310 (x4)
System Memory	8GB
Hard Disk Capacity	160GB
Allocated Memory Bandwidth (Huge Pages)	2805MB/s
Allocated Memory Bandwidth	2690MB/s
Unallocated Memory Bandwidth (Huge Pages)	1205MB/s
Unallocated Memory Bandwidth	747MB/s
Bandwidth to Hard Disk Cache	1019.23MB/s
Bandwidth to Hard Disk	60.36MB/s
Average Hard Disk Seek Time	8.1ms

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/time.h>

#define MT_ALLOCATE 1024*1024*1024

int main(int argv, char** argc) {
  char *arr = malloc(MT_ALLOCATE );

  struct timeval starttime;
  struct timeval endtime;
  double timedelta;

  gettimeofday(&starttime, NULL);
  memset(arr, 0, MT_ALLOCATE);
  gettimeofday(&endtime, NULL);
  timedelta = (endtime.tv_sec - starttime.tv_sec) * 1.0 +
              (endtime.tv_usec - starttime.tv_usec) / 1000000.0;
  printf("Elapsed time: %fs\n", timedelta);

  gettimeofday(&starttime, NULL);
  memset(arr, 0, MT_ALLOCATE);
  gettimeofday(&endtime, NULL);
  timedelta = (endtime.tv_sec - starttime.tv_sec) * 1.0 +
              (endtime.tv_usec - starttime.tv_usec) / 1000000.0;
  printf("Elapsed time: %fs\n", timedelta);

  return 0;
}
```

Figure 6-3: Buffer-Write Benchmark

```c
#include <stdio.h>

int main() {
   FILE * fp = fopen("big.dat", "rb");
   int i, res;
   char c;
   for (i = 0; i < 10000; i++) {
      fseek(fp, rand()%1000000000, SEEK_SET);
      res = fread(&c, 1, 1, fp);
   }
}
```

Figure 6-4: Random Disk IO Benchmark

transfer rates of 747MB/s and 2690MB/s, respectively, per Figure 6.1.

This test was run using the default Linux memory-allocation mechanism. For comparison, this program was linked against the "libhugetlbfs" library, a library that overrides the default behavior of malloc() under Linux and causes it to allocate "huge pages", single pages that are larger than the default page size. On the x86_64 platform, the only available huge page size is a 2MB page. (Regular pages are 4KB.) Because fewer huge pages are required to fill a region in memory, it requires less CPU time to construct a page table containing all the necessary entries. With huge pages, performance of a newly-allocated buffer increased substantially. The first call to memset() took 0.83s. The second call was slightly faster as well, at 0.36s.

A second micro-benchmark determined the maximum disk throughput of the hard disk used for the tests. This test was performed using the "hdparm" command-line utility. The test indicated a 1019.23MB/s for cached reads, and 60.36MB/s for reads from the physical disk 6.1.

A third micro-benchmark determined the average time required for the disk to seek to a new location on disk. This test was performed by creating a 1GB data file named "big.dat" filled with zero bytes, and compiling and executing the code in Figure 6-4, which performs 10,000 random seek operations in that data file and

Table 6.2: Compression Micro-Benchmark
The listed compressors were tested on a 512MB string consisting of the character '0' and a series of random letters, in the proportions indicated. In each case, the compressor was run 10 times on the exact same data; the average time of a single run is therefore equal to the given times divided by 10.

	Lempel-Ziv		Run-Length Encoding	
Ratio	Time	Space	Time	Space
$\frac{3}{4}/\frac{1}{4}$	83.29s	136687429B	97.22s	658174340B
$\frac{7}{8}/\frac{1}{8}$	60.44s	69396924B	67.46s	329088295B
1/0	37.78s	2105388B	37.63s	5B

then exist. Prior to executing the code, the operating system's buffer pool was cleared, to prevent the random reads from being taken from system memory. The code executed in 80.982s. This leads to an average seek time of 8.1ms per seek.

The LempelZiv and Run-Length Encoding compressors were also compared in a series of microbenchmarks to see how they would deal with simple data streams. A 512MB buffer was allocated, and split into two regions. The first region was filled with the character '0'; the second was filled with a random string of capital and lower-case alphabet characters. Several splits between these two regions were tested; $\frac{3}{4}$ / $\frac{1}{4}$, $\frac{7}{8}$ / $\frac{1}{8}$, and all zeros. (Run-length encoding produced output that was considerably larger than the input string for any split closer to even.) The results can be seen in Table 6.2.

It's clear from the results in this table that run-length encoding is considerably less space-efficient, and considerably slower, than Lempel-Ziv encoding for more-random data sets. However, if a data set is highly uniform, run-length encoding can produce vastly-smaller output than Lempel-Ziv encoding.

The viability of Hilbert curves as a compression mechanism was also considered. A collection of images were each serialized along a Hilbert curve, as shown in Figures 6-5 and 6-6, using the Hilbert-curve-generation code shown in Figure 6-7. The image data was then compressed using the "gzip" library [10], an implementa-

Table 6.3: Hilbert-Lempel-Ziv Compression Results

Image ID	Raw file size	HLZ size	Linear size	$\frac{\text{HLZ}}{\text{Linear}}$
1003507_47622857	768kb	416kb	416kb	100%
1148704_62052785	12288kb	3696kb	3584kb	103%
239732_1868	3072kb	1936kb	2020kb	96%
352029_9678	768kb	528kb	548kb	97%
394492_2293	3072kb	680kb	732kb	93%
431419_30950322	3072kb	2576kb	2772kb	93%
489025_77341890	3072kb	1604kb	1920kb	84%
559699_68176265	3072kb	1612kb	1804kb	89%
777635_11317903	3072kb	1840kb	1956kb	94%

tion of a Lempel-Ziv-type compression algorithm. The results of this compression are shown in Table 6.3. As can be seen, Hilbert curves did almost uniformly help with compression, but only by about 5%.

6.4 Rough Pass

For the rough pass, only the NOAA RTMA data set was considered. The NOAA data set contains a variety of different matrices of different types, in terms of dense and sparse differences and amounts of noise in the data. It is also a relatively small data set; not so small as to be unrealistic, but small enough to allow an array of tests to be executed quickly.

The test procedure was to import the data into the versioned storage system once per test, applying any compression or storage techniques being used for the test; then run a weighted-random series of "Select" operations against the data. The data consists of multiple matrices per time step, each representing a different type of measurement; each type was assigned its own versioned-matrix object. In total there were 88 versioned matrix objects.

Because the data set was smaller than system memory, the operating system buffer pool was cleared prior to each import and prior to each Select operation, in order to force data to be read from disk.

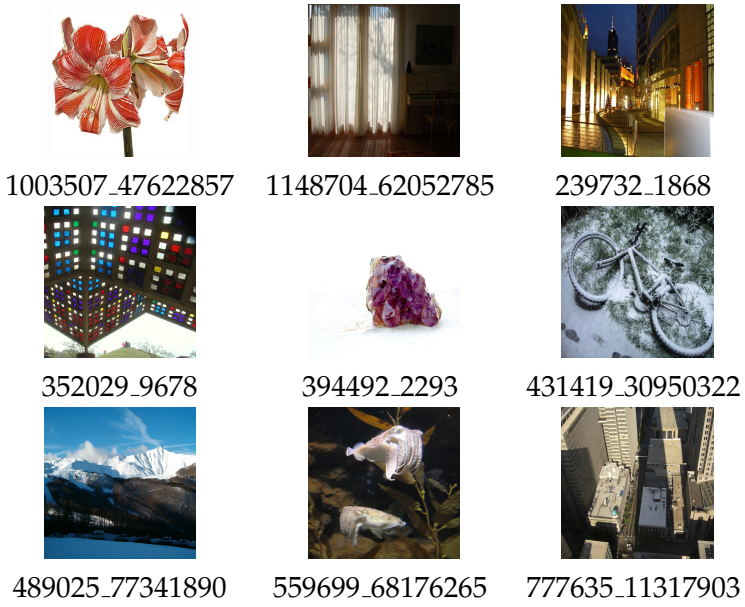


Figure 6-5: Hilbert-Curve Test Images
Nine randomly-selected images from Stock.XCHNG[7], used for encoding testing. Images are identified by their Stock.XCHNG ID numbers.

For the "Select" operations, a series of queries were run, each querying for the data of one complete version. Chunking and colocation are tested independently at this stage, and none of the proposed compressors are designed to perform appreciably differently in the presence of a subset query than a query for the whole data set, so selecting a subset of a matrix was not anticipated to make any difference. The second pass of experiments validates this hypothesis.

For each Select, a versioned matrix was first chosen at random. Within that versioned matrix, 80% of the time, the most recent version would be selected, and 20% of the time, an older version would be chosen at random and selected. The intent was to simulate a mixed workload, where one set of users is regularly accessing the most-recent data for immediate or cumulative analysis, and another set is referring to past versions for broader analysis or to view particular data sets that might be referenced in papers or other documents.

For tests that involve differencing, a "reverse delta" approach was used for delta ordering: The most-recent version in a version chain was always material-

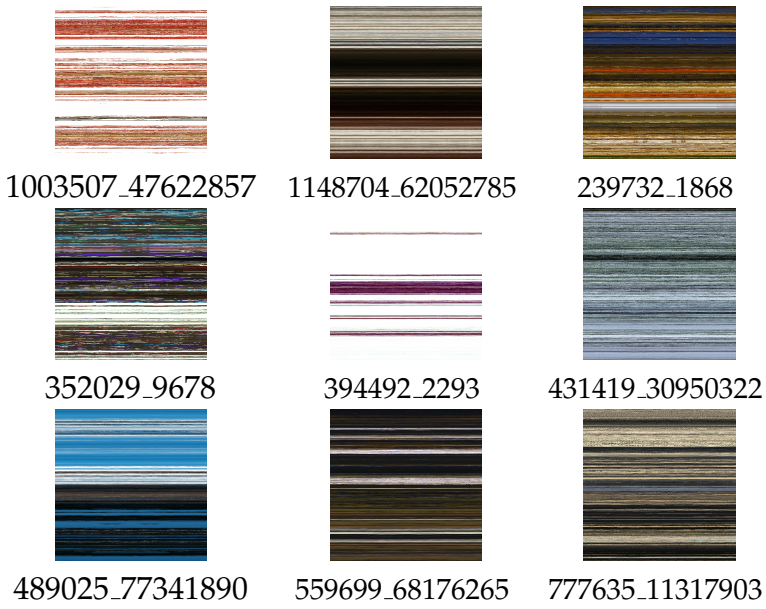


Figure 6-6: Hilbert-Compressed Images
Images from Figure 6-5 after having been run through the Hilbert-curve phase of the algorithm. The linear data was re-wrapped in row-major order to the original square image shape for display.

ized, and other versions in the chain were differenced against it and each other in reverse-chronological order. For example, in a chain of length 3, version 3 would be materialized, version 2 would be differenced against version 3, and version 1 would be differenced against version 2.

In all cases, if storing a delta or a compressed object used more disk space than storing the corresponding matrix in uncompressed form, the matrix was stored in uncompressed form.

6.4.1 Compression Algorithms

Several available compression algorithms were also considered. Tests were run using both compression algorithms, and without compression for comparison. The results are shown in Tables 6.4 and 6.5. The former table lists the results of an array of compressors against arrays that have previously been differenced with the hybrid differencing algorithm. The latter shows a small number of algorithms op-

```
from math import sin, cos, radians
from PIL import Image
import struct
import gzip
import bz2
import scipy

def hilbert(size_of_side):
    curr_angle = [0]
    curr_coord = [0,0]
    size = 1

    def right(angle):
        curr_angle[0] += angle
    def left(angle):
        curr_angle[0] -= angle
    def forward(size):
        curr_coord[0] += int(round(sin(radians(curr_angle[0])))) * size
        curr_coord[1] += int(round(cos(radians(curr_angle[0])))) * size
        return tuple(curr_coord)

    def hilbert_gen(level, angle):
        if level == 0:
            return
        right(angle)
        for x in hilbert_gen(level - 1, -angle):
            yield x
        yield forward(size)
        left(angle)
        for x in hilbert_gen(level - 1, angle):
            yield x
        yield forward(size)
        for x in hilbert_gen(level - 1, angle):
            yield x
        left(angle)
        yield forward(size)
        for x in hilbert_gen(level - 1, -angle):
            yield x
        right(angle)

    def round_size_pow_2(size):
        size -= 1
        counter = 0
        while size > 0:
            size = size >> 1
            counter += 1
        return counter

    yield (0,0)
    for x in hilbert_gen(round_size_pow_2(size_of_side), 90):
        yield x
```

Figure 6-7: Hilbert-curve-generation code

Table 6.4: Compression-Algorithm Performance on Un-Differenced Arrays

Compression	Size	Query Time	Number Compressed
Uncompressed	253M	2.75s	N/A (of 88)
Lempel-Ziv	124M	5.47s	88 (of 88)
Run-Length Encoding	240M	3.82s	15 (of 88)
De-Duplication	226M	2.75s	23 (of 88)

Table 6.5: Compression-Algorithm Performance on Delta Arrays

Compression	Size	Query Time	Number Compressed (un-delta'ed + delta'ed)
Hybrid Delta only	133M	3.53s	N/A+79 (of 88)
Lempel-Ziv	94M	4.01s	9+55 (of 88)
Background Subtraction	131M	3.68s	1+7 (of 88)
Run-Length Encoding	133M	3.32s	0+0 (of 88)
Adaptive Huffman	96M	26.29s	9+56 (of 88)
Bitmap Encoding	133M	4.53s	0+0 (of 88)
PNG compression	116M	5.93s	0+39 (of 88)
JPEG 2000 compression	118M	20.23s	0+39 (of 88)

erating on their own: The Lempel-Ziv algorithm to demonstrate that it does not perform as well on its own as when used in conjunction with the hybrid differencing algorithm, and Run-Length Encoding because it will tend to work well when stored matrices are sparse, so it may work more efficiently on its own then after arrays have already been differenced.

For the first comparison, the JPEG 2000 and Adaptive Huffman encodings both produce relatively small files, but with a very high cost in terms of query performance. Both are expensive algorithms. Additionally, they do not produce the smallest files; simple Lempel-Ziv compression performs better.

It should be noted that the implementation of JPEG 2000 used here was only able to compress 16-bit and 8-bit dense arrays; more-sophisticated implementations might achieve better compression. Note that JPEG 2000 was only able to compress just under half of its input files. The PNG implementation used here had a similar limitation, and was able to compress the same number of files, though slightly more effectively and a great deal more quickly.

Background Subtraction barely performed better than straight delta compression, and Run-Length Encoding and Bitmap Encoding did not perform better than straight delta compression.

Lempel-Ziv achieved the smallest file size, and ran as quickly as any algorithm that was able to achieve more than 2% compression. It would seem to be clearly superior to the other algorithms used here for any use case that doesn't consider performance to be absolutely essential.

For the second comparison, as can be seen, both compression algorithms performed more slowly than the uncompressed data. Per our cost model, this makes sense: These compression algorithms are not highly-tuned algorithms, and the API to access the algorithms is not particularly optimized either; decompressing a matrix requires a substantial amount of memory copying and allocation.

However, Run-Length Encoding saves relatively little disk space for its query-

Table 6.6: Performance of Selected Differencing Algorithms

	Time			Number
Deltaer	Import	Query	Size	Compressed
Uncompressed	4.31s	2.75s	253M	N/A (of 88)
Dense	8.99s	3.41s	168M	48 (of 88)
Sparse	21.15s	3.21s	191M	22 (of 88)
Hybrid	15.16s	2.81s	142M	57 (of 88)
MPEG-2-like Matcher	9598.1	39.60s	138M	79 (of 88)
BSDiff	343.8s	3.59s	133M	79 (of 88)

runtime cost. Also, only 15 of the 88 matrices in the data set could be stored using less space with Run-Length Encoding than without it; the remaining 73 matrices were stored uncompressed regardless. Therefore, Run-Length Encoding does not seem to make sense as a general-purpose algorithm for these data sets.

Lempel-Ziv encoding, on the other hand, achieves substantial disk-space savings; it cuts disk usage by around 50%. It seems to be an effective trade-off between performance and disk usage. As a result, it does seem worth considering as an algorithm.

The second comparison also tested de-duplication. De-duplication does not have any impact on query performance, at least not in the absence of a buffer pool, because it does not change the size or content of any array object; it simply prevents identical array objects from using additional space on disk. De-duplication proved to be more effective than run-length encoding at reducing the total data size, though not nearly as effective as Lempel-Ziv encoding.

6.4.2 Delta Algorithms

A variety of delta algorithms have been considered thus far. Their implementations are compared in Table 6.6.

One immediate observation is that both the MPEG-2-like spatial matching algorithm and the BSDiff algorithm are much more expensive to run than are the remaining differencing algorithms. The MPEG-2-like implementation in particular

is orders of magnitude slower than any other option, owing to the huge number of comparisons that it performs. They do produce reasonably small differences, smaller than those of any other proposed algorithm. However, the disk-space savings are not proportional to the increased algorithm runtime.

Of the remaining implementations, the dense differencer has the fastest runtime. This is in part because of the implementation of the delta algorithms: The array library that is used here to represent sparse arrays always copies the result of the addition into a new array. When adding two sparse arrays, this is often the optimal approach, particularly if the sparse arrays have few common matrix cells populated. However, when adding a sparse array to a dense array, the optimal approach would be to update the values of the dense array in place. This results in considerable additional memory I/O, which per the cost model would impose a performance penalty. The difference should be relatively small, however; per our model, on the test system, the extra memory copy should add about 10% to the cost of reassembling the delta, which is itself only one part of the overall query process.

The hybrid algorithm both uses the least disk space and yields the best query performance. It also is able to usefully compress more arrays than either of the dense or sparse algorithms. Its import time is considerably longer than that of the dense algorithm, but still considerably shorter than the sparse approach. Because it achieves good performance and efficient disk storage, the hybrid algorithm appears to be a good general-purpose differencing algorithm for matrices.

It should be noted that all compression algorithms, including the hybrid algorithm, were slower than the uncompressed operation, despite having considerably less total data to read. No additional data was being read from disk; however, a substantial amount of time was being spent copying data between large memory buffers. NumPy in particular tends to create a new array for each arithmetic operation rather than mutating one of the operands in place to produce the new array, though it can be explicitly instructed to not do this. A number of optimizations were introduced for the next round of tests, as discussed in Section 6.5; future re-

sults, such as those in Table 6.11, do indicate that the hybrid algorithm can be faster than uncompressed operations.

6.4.3 Compressing Delta Output

Delta compression and traditional compression are not mutually exclusive. The first two rows in Table 6.5 consider delta compression with and without Lempel-Ziv compression on top of it. As can be seen, the addition of Lempel-Ziv compression increases the query-execution time by about 15%, but decreases the required storage space by about 25%. These percentages are close enough that different applications would likely choose to make different trade-offs between them.

6.4.4 Subversion and Git

This matrix-storage system is a versioning system; therefore, it makes some sense to compare it against other contemporary versioning systems.

For this test, a differencer was used that combines the proposed hybrid-delta compression with Lempel-Ziv compression: hybrid deltas are created, and each array in the hybrid delta is compressed using Lempel-Ziv compression.

In both Subversion and Git, each timestamp was mapped to a commit and each matrix was mapped to a version file, so a matrix version was a file at a particular commit. The Subversion repository was manipulated using the "pysvn" wrapper for the Subversion library API, with some calls to the Subversion command-line tools for operations not available through the library or wrapper. At the time of this writing, Git did not have a library API available; Git operations were implemented by calling out to the Git command-line tools.

Note that, in both cases, arrays needed to be retrieved from storage and written to disk before they could be read by the versioning API; versions could not be read directly into memory. This extra disk write is a design constraint of the current Subversion and Git interfaces, but it does decrease performance and increase disk usage.

Table 6.7: Subversion and Git Differencing Performance

Differencing Method	Import Time	Query Time	Data Size
Uncompressed	4.31s	2.75s	253M
Hybrid Delta with Lempel-Ziv	13.1s	5.47s	90M
Subversion	47.0s	7.97s	111M
Git	100.5s	3.70s	147M

Subversion compresses its data automatically. To ensure optimal disk usage, however, the "svnadmin pack" command was run after loading all data into the repository. Git does not compress its data automatically; the "git repack" command was used after loading all data in the repository to compress it. "git repack" accepts a variety of arguments to tune its operation; none were passed in, so it operated per its defaults.

The results of this comparison can be seen in Table 6.7. Of immediate note, the hybrid delta compression implementation used considerably less disk space than either Subversion or Git. It also performed uniformly faster than Subversion; it took less than $\frac{1}{3}$ as long to import data, and about $\frac{2}{3}$ as long to execute the query workload.

Git executed the query workload considerably faster than either Subversion or the hybrid delta compression. However, it took almost 8 times as long as hybrid delta compression to complete its compression pass. This is interesting because Git is the only parallel compression implementation; it used all 16 CPU cores available on the test system, while the other compression algorithms used a single core. The reason for this appears to be a memory constraint: Even though the data set itself is only 253 megabytes, during this test Git was observed to have exceeded the available 8 gigabytes of system memory; it was swapping heavily. This amount of memory consumption could prove problematic for many use cases where data sets are large as compared to system memory.

Based on these numbers, the hybrid delta compression appears to be superior in most dimensions to both Subversion and Git, for this type of matrix delta com-

Table 6.8: Workload Query Time with Chunking, Colocation

	Chunking	No Chunking
Colocation	2.61s	2.58
No Colocation	2.80s	2.94s

pression. This is somewhat unsurprising; both were designed for storing small text files rather than large binary arrays.

6.4.5 Chunking and Colocation

Neither chunking nor colocation compresses data, but both can potentially have an impact on query performance, by reducing the number of disk seeks needed to access a particular piece of data and reducing the amount of extraneous data that must be read in order to get at the desired part of a matrix.

Chunking and colocation were tested on the NOAA weather dataset, to see what impact they have on performance. The standard query workload was replaced with the "Range" query workload discussed under "Data Variants" below, in order to make better use of collocated matrices. The results can be seen in Figure 6.8. No queries selected a fraction of a matrix.

As can be seen, the performance difference between any of these options is small. Colocation shaves about a tenth of a second off of the query runtime; chunking adds about a tenth of a second to it.

Chunking was analyzed further in Tables 6.10, 6.11, and 6.12. The range-select operations in these tables represent a query of a single chunk. The tables have chunk sizes of 1MB, 10MB, and 100MB, respectively. As can be seen, the query cost goes up roughly proportionally to the size of the chunk.

6.4.6 Space- and Performance-Optimized Deltas

A variety of possible heuristic and algorithmic approaches have been discussed to find better orderings for delta chains or delta trees. We assume here that no

Table 6.9: Optimal Delta Order, Heuristic Approach

Compression	Size	Time		Number
		Compression	Query	Compressed
Performance-Optimized	374M	42.45s	4.13s	407 (of 420)
Space-Optimized	383M	102.43s	3.79s	388 (of 420)
Linear (not optimal)	368M	5.99s	3.53s	411 (of 420)

algorithm with an exponential runtime is acceptable in the general case. There may be use cases where such algorithms are desirable or even necessary, but in general, they tend to become unacceptably expensive very quickly as data size grows.

Instead, the polynomial-time approaches discussed earlier in the paper have been implemented and tested. The heuristics use both a full $O(N^2)$ comparison of each array against each other array, and an $O(N)$ heuristic that involves down-sampling each array to a 16x16 smaller matrix, and comparing the smaller matrices against each other.

On the test data set, with the $O(N^2)$ operation, both algorithms ran for over 200 seconds.

With the heuristic approach, the results can be seen in Table 6.9.

As can be seen, even the polynomial-time heuristics are rather expensive as compared to their linear-time delta-chain counterparts. Note that these algorithms have been implemented in a parallelizable way; they are able to take advantage of all 16 CPU cores on the test system, rather than just one. This gives these algorithms a factor-of-16 performance advantage over their counterparts. The down-sampling implementations are considerably cheaper, but still take much longer than their linear counterparts to run.

Comparing the results in each category: For disk space, the performance-optimized heuristic uses about 2% more space than the simple linear approach, which is potentially quite reasonable; but the space-optimized heuristic uses about 5% more space, which is less reasonable. The compression times have a huge difference; dis-

counting the extra factor of 16 due to parallelism, by the wall clock the performance-optimized algorithm ran 7 times as long as the linear one and the space-optimized algorithm ran 17 times as long. In terms of query performance, the performance-optimized approach was slowest, at 17% longer than linear.

The linear compression algorithm is able to make a single long delta chain very easily. It is given arrays that tend to difference against each other effectively without forming any loops. Any approach to compression must avoid producing loops; these heuristic approaches tend to do so by creating multiple shorter chains, which will tend to use more disk space. The need to read additional data from disk will tend to drive up query costs.

Additionally, delta ordering can be important: If a user requests a series of matrices in an order that is the same order as the delta chain, our current implementation is able to use something approximating an iterator model: Return each matrix as it is available. If a different order is requested, however, the entire matrix must be reassembled in main memory; a process that involves additional memory-buffer copies and work in general to find the optimal order for assembling all of the pieces. The space-optimized algorithm in particular tends to not group or order matrices for differencing at all; it is entirely capable of turning the delta chain into a wide delta tree; and this can cause some further performance hinderances.

Based on these results, it seems clear that these "near-optimal" heuristics are not worthwhile. To the extent that either algorithm worked, it seemed that the performance-optimized algorithm produced the slower delta plan, and the space-optimized algorithm produced the larger data set. It seems that following the version tree and simply differencing in chronological order leads to better results. The current test implementation does test arrays in version order.

6.5 Detailed Analysis

Based on the results in the previous section, it seems that the best general-purpose data-storage approach is to use chunked LempelZiv-compressed delta compres-

sion with the hybrid algorithm and with simple delta chains in version order. Several variants of this algorithm are now tested in more depth on a wider variety of data sets.

Based on past analyses, a number of optimizations are introduced to the algorithms tested here. Notably, the implementation library is further optimized to minimize copies into newly-allocated memory buffers at the expense of some flexibility in terms of easily supporting new library features. Additionally, x86_64 Huge Pages are used for all in-memory storage of matrices. Based on previous test results, huge pages perform considerably better for copying data into new buffers.

6.5.1 Algorithm Variants

Based on the above testing, the most promising compression methods seem to revolve around hybrid-delta and Lempel-Ziv compression. A number of trials have been run using various arrangements of these two compression methods, on the Open Street Maps data set. The results of these trials can be seen in Tables 6.10, 6.11, and 6.12.

The test permuted several variables:

1. **Chunking** - The array was split into smaller arrays (1MB, 10MB, or 100MB, as indicated on each table) prior to differencing and compression
2. **Colocation** - Each chunk's full history of versions was stored in a single file, rather than storing each version of each chunk in a separate file
3. **Deltas** - The array was compressed using hybrid delta compression
4. **Materialization** (abbrev. **"Mat'zn"**) - After being delta-compressed, matrices were re-materialized in an even distribution across the set until 10% of the disk-space savings from delta compression were negated by re-materializing matrices
5. **LempelZiv** - Matrix objects, or delta objects where applicable, were compressed with Lempel-Ziv compression

There was also a "Basic Uncompressed" variant, that lacked any compression or chunking. It was included for reference.

The test also contained a number of different Select operations:

1. **1-Array Select** - The most-recent stored array was selected.
2. **1-Array Range Select** - The range consisting of four "pixels", ie., 12 total values, at the origin was selected. The aim was to retrieve a single chunk.
3. **16-Array Select** - All 16 stored arrays were selected, in order from most-recent to least-recent (so in the order of differencing, starting with the materialized array).
4. **16-Array Range Select** - A region consisting of 12 values per array across all 16 arrays was selected.

This test was run using x86_64 huge pages, to increase memory-buffer-copy performance. Timing was wall-clock time; the test was run while the computer was otherwise idle. Disk bytes-read counts were as provided by the "read_bytes" field Linux's /proc/PID/io virtual filesystem; it is a count of bytes read directly from the physical disk (or from its small onboard buffer cache), excluding I/O requests that hit the operating system's buffer pool. In order to minimize tests' effect on each other, the operating system's buffer pool was flushed prior to each operation.

With these tests, one can immediately see that chunking has substantial overhead associated with it. Comparing basic chunking with basic uncompressed, for 1MB chunks, the 1-Array Select performed twice as slowly with chunking as without; 19.19s vs 38.64s. For 10MB chunks, the 1-Array Select performed roughly 1.5x as slowly; 19.20s vs 27.38s. The 16-Array Select's performed comparably, just scaled by a factor of 16.

Much of this overhead is due to the implementation of chunking: In order to potentially support different chunking schemes in the future, the chunk layout of

a versioned array was stored in its own array, indicating the starting coordinate of each chunk. This array needed to be read and scanned prior to executing any query against the data. Additionally, because these queries return complete in-memory arrays, chunking necessitates some additional in-memory copying; each chunk must be copied into the appropriate region of the larger array prior to returning the larger array. Particularly for the 1MB chunks, disk-seek times also play a role.

These results also demonstrate the effectiveness of chunking for selecting small regions of a larger array. Query time for these small Selects is approximately proportional to chunk size, even down to 1MB chunks where disk-seek time is starting to become relevant. For example, the 16-Array Range Select took 6.25s with 1MB chunking, 14.17s or 10MB chunking, and around 260s for the two runs with no chunking.

Colocation, on the other hand, has no benefit at all; in fact, its overhead gives it a substantial performance penalty. For example, "Chunking with Colocation, Deltas" runs in 58.86s with 1MB chunks, and "Chunking with Deltas" runs in 31.67s. Part of this is because of the caching scheme used by this implementation: This code does not have a buffer pool per se. However, in order to efficiently reassemble delta chains, it must keep the most-recently-used matrix in memory so that when the next matrix is requested by the code that assembles the query result, the entire rest of the chain need not be reconstructed. When reassembling chunked arrays, it's necessary to store the most-recent version of each chunk because chunked arrays are assembled piecewise. When the chunks are stored collocated with other arrays, this caching mechanism may store duplicates of the entire chunk-file rather than of each array, since that is the object that was requested from disk.

With regard to memory usage, when compared to the other results, the 100MB-chunk results in Figure 6.12 seem somewhat odd. Several of the timing numbers are much larger than one might expect, as are the numbers for data read from disk. Additionally, some rows are missing. Looking at the 10MB-chunk numbers,

it appears that a larger chunk size might not be particularly beneficial; the range-select queries were already considerably slower with 10MB chunking than with 1MB, and the remaining queries were not appreciably slower. The inclusion of the 100MB-chunk data was deliberate: The particular chunking implementation in that test has a memory leak. The leak was initially accidental, but it seemed representative of a common buffer-pool scenario: Even though this test does not implement retrieving arrays out of a buffer pool, this test did contain a small amount of buffer-pool code as part of an experiment, and arrays were not flushed properly from this unused buffer pool. Ordinarily, this has negligible cost; a few small unused buffers sitting idle in memory. However, with 100MB chunks, these buffers aren't so small, and with collocation, up to 16 of them can be stored together, for a 1.6GB file. A few of these stayed in memory, and while reading large chunks a substantial amount of additional information was pulled into memory. Ultimately, the test exhausted all available physical and virtual memory and terminated with an error. The unexpected additional disk I/O is swap activity. This benchmark is included for the consideration of future buffer-pool designers, as a bit of a warning that large matrices can very easily grow to exceed system memory.

With regard to disk usage, as expected, the combination of Lempel-Ziv and delta compression consistently yields the smallest on-disk footprint. Deltas and Lempel-Ziv alone are each relatively effective. With 10MB chunks, for example, "Chunking with Deltas" completed the 16-Array Select in 249.80s reading 2GB from disk. "Chunking with Deltas+Mat'zn, LempelZiv" completed the 16-Array Select in 335.22s reading around 1.8GB from disk. With no compression, the operation took 289.16 seconds and read 15GB from disk. Note, however, that both of these implementations used chunking; Basic Chunking also read 15GB from disk but took 451.01s. The combination of Lempel-Ziv and hybrid delta compression were therefore clearly effective for both increasing performance and decreasing disk usage.

For this data set and query workload, materialization seems to have a relatively small effect. This is most likely largely because the query workload never forces

the assembly of a long delta chain just to get at the array at the end of the chain.

Overall, these results seem to indicate that, for the given query workload, Lempel-Ziv compression combined with hybrid deltas lead to both decreased disk usage and increased query performance. Chunking helped substantially for range queries. Other techniques did not seem to be substantially effective.

A similar test was also run on a different test system, one with otherwise-similar performance except that its hard-disk storage was a RAID array that was measured as reading 220MB/s using the same mechanism by which Caneland's was measured to achieve 60MB/s. The test system in question also had access to a slightly different set of versioned tiles; it included tiles from the start of 2010 as well as 2009, skipping several versions at the start of 2010 so that those versions did not difference against each other quite as cleanly as those in other tests. The results of this operation can be seen in Table 6.13. As can be seen, in this environment, much of the performance advantage of the differencing and compression algorithms is negated. For example, the 22-Array Select ran in 342s with differencing and Lempel-Ziv compression, as compared to 359s for basic chunking alone, despite using 5 times fewer bytes on disk. If hard disk performance increases relative to CPU and memory performance, the performance advantages of differencing algorithms are reduced.

Table 6.10: Open Street Maps data, 1MB Chunks

	1-Array Select		1-array Range Select		16-Array Select		16-Array Range Select		Total Bytes	Number
Compression	Bytes Read	Time	Bytes Read	Time	Bytes Read	Time	Bytes Read	Time	On Disk	Of Files
Chunking with Colocation, Deltas+Mat'zn	1.08GB	66.55	3.41MB	0.41	1.11GB	302.52	1.50MB	4.83	2.11GB	362
Chunking with Deltas	1.00GB	31.67	3.48MB	0.35	2.02GB	295.88	6.12MB	4.63	3.00GB	5792
Chunking with Colocation, LempelZiv	0.212GB	76.32	0.72MB	0.44	1.80GB	386.24	5.25MB	4.61	1.19GB	362
Basic Chunking	1.00GB	38.64	3.47MB	0.37	1.50GB	538.41	46.94MB	6.25	16.00GB	5792
Chunking with Deltas+Mat'zn, LempelZiv	0.122GB	18.71	0.63MB	0.35	1.81GB	371.17	6.84MB	5.25	1.91GB	5792
Basic Uncompressed	1.00GB	19.19	1.00GB	18.42	15.00GB	255.97	15.00GB	253.06	16.00GB	16
Chunking with Colocation, Deltas	1.08GB	58.86	3.42MB	0.42	2.01GB	308.11	4.40MB	4.35	3.00GB	362
Chunking with LempelZiv	0.122GB	20.20	0.61MB	0.30	1.81GB	379.13	6.72MB	4.78	1.91GB	5792

Table 6.11: Open Street Maps data, 10MB Chunks

	1-Array Select		1-array Range Select		16-Array Select		16-Array Range Select		Total Bytes	Number
Compression	Bytes Read	Time	Bytes Read	Time	Bytes Read	Time	Bytes Read	Time	On Disk	Of Files
Chunking with Colocation, Deltas+Mat'zn	1.01GB	36.45	30.0MB	1.02	1.10GB	240.92	13.0MB	6.20	2.10GB	37
Chunking with Deltas	1.53GB	42.63	30.2MB	0.96	2.00GB	249.80	42.5MB	6.86	3.00GB	592
Chunking with Colocation, LempelZiv	1.36GB	25.14	3.07MB	0.64	0.19GB	319.18	39.0MB	9.00	2.01GB	37
Basic Chunking	1.00GB	27.38	30.2MB	1.06	15.00GB	451.01	450MB	14.17	16.00GB	592
Chunking with Deltas+Mat'zn, LempelZiv	0.13GB	18.63	2.90MB	0.61	1.89GB	335.22	39.5MB	10.32	2.01GB	592
Basic Uncompressed	1.00GB	19.20	1.00GB	19.65	15.00GB	289.16	15.0GB	276.18	16.00GB	16
Chunking with Colocation, Deltas	1.01GB	34.87	30.6MB	1.15	2.00GB	257.61	42.1MB	6.78	3.00GB	37
Chunking with LempelZiv	1.27GB	18.93	2.76MB	0.62	1.88GB	331.53	39.3MB	9.69	2.01GB	592

Table 6.12: Open Street Maps data, 100MB Chunks

	1-Array Select		1-array Range Select		16-Array Select		16-Array Range Select		Total Bytes	Number
Compression	Bytes Read	Time	Bytes Read	Time	Bytes Read	Time	Bytes Read	Time	On Disk	Of Files
Chunking with Deltas	2.58GB	528.58	300MB	8.93	2.80GB	545.41	467MB	67.85	3.00GB	80
Chunking with Colocation, LempelZiv	2.32GB	272.89	796MB	208.65	38.6GB	5210.08	879MB	135.76	2.11GB	5
Chunking with Deltas+Mat'zn, LempelZiv	0.393GB	115.20	369MB	91.86	9.33GB	1498.29	998MB	168.44	2.11GB	80
Basic Chunking	1.00GB	38.66	300MB	10.93	15.00GB	547.14	4.50GB	173.19	16.00GB	80
Basic Uncompressed	1.00GB	22.77	1.000GB	22.58	15.00GB	345.55	15.00GB	342.45	16.00GB	16
Chunking with LempelZiv	0.133GB	19.48	52MB	8.12	3.92GB	511.92	814MB	155.83	2.11GB	80

Table 6.13: Open Street Maps data, 10MB Chunks, Different Test System

	1-Array Select		1-array Range Select		22-Array Select		22-Array Range Select		Total Bytes	Number
Compression	Bytes Read	Time	Bytes Read	Time	Bytes Read	Time	Bytes Read	Time	On Disk	Of Files
Chunking with Colocation, Deltas+Mat'zn	0.98GB	20.58	31.7MB	0.55	3.47GB	341.79	43.7MB	8.78	4.57MB	74
Chunking with Deltas	1.00GB	15.19	30.8MB	0.49	3.32GB	324.22	46.5MB	9.97	4.34MB	814
Chunking with Colocation, LempelZiv	0.13GB	20.40	3.89MB	0.50	2.19GB	369.19	29.0MB	10.96	2.56MB	74
Basic Chunking	0	10.51	30.5MB	0.49	20.0GB	358.85	601MB	10.82	22.0GB	814
Chunking with Deltas+Mat'zn, LempelZiv	0	14.99	0.80MB	0.49	2.31GB	356.18	26.1MB	10.23	2.56GB	814
Basic Uncompressed	0	0.73	0	0.65	21.0GB	120.36	21GB	120.63	22.0GB	22
Chunking with Colocation, Deltas	0.93GB	20.26	32.0MB	0.51	3.27GB	314.34	46.4MB	9.36	4.34GB	74
Chunking with LempelZiv	9MB	15.89	0.75MB	0.51	2.31GB	373.96	26.0MB	10.89	2.56GB	814

6.5.2 Data Variants

The proposed chunked delta algorithm has been run against the NOAA weather data, the ConceptNet data, and the OpenStreetMaps map data, with a query workload intended to simulate normal use. The full contents of all data sets were used. This test was executed exclusively using the hybrid differencing algorithm, to verify its effectiveness on an array of different types of data.

The query workload consists of four types of queries:

1. **Most-Recent Matrix** - The "head", or most-recent version, of the matrix, is always selected
2. **Random Single Matrix** - A version is chosen at random and selected
3. **Random Range** - A version is chosen uniformly at random, and a range is selected around that version with a Gaussean distribution. Every version within the range is selected sequentially.
4. **Random Modification** - The most-recent version of a matrix is selected. A small number of random-noise values between 0 and 100 are added to cells of the matrix, and the modified matrix is saved as a new version.

The intent of these types of operations is to simulate a general-purpose query workload, with some users selecting individual arrays, some users selecting only parts of arrays, some users selecting multiple arrays at once, and some users or scripts generating and inserting new arrays to be selected from. It's possible that any given real-world application will have any one of these four operations in a much greater proportion than the other three. Therefore, these basic query types are combined into five types of query workloads. Each of these workloads operates on a single versioned matrix, selected uniformly at random.

1. **Head** - The most-recent matrix version is selected with 90% probability; a random single matrix is selected with 10% probability. This is repeated 30 times.

Table 6.14: Various Workload Tests, Hybrid Delta Compression

Data Set	Compressor	Data Size	Time To Compress	Query Workloads (Time to Execute)				
				Head	Random	Range	Insert	Everything
NOAA Weather Data	Hybrid, LZ	2.3G	663.12s	11.48s	6.76s	7.41s	2.27s	104.94s
NOAA Weather Data	None	8.2G	N/A	0.232s	2.26s	5.56s	0.43s	30.48s
ConceptNet	Hybrid, LZ	11M	11.37s	6.88s	80.73s	70.11s	0.18s	17.62s
ConceptNet	None	728M	N/A	0.43s	2.18s	3.31s	0.18s	1.78s

2. **Random** - A random single matrix is selected. This is repeated 30 times.

3. **Range** - With 10% probability, a random single matrix is selected. With 90% probability, a random range with a standard deviation of 10 is selected. This is repeated 30 times.

4. **Everything** - A query is chosen from the four query types with equal probability. If a random range query is selected, it has a standard deviation of 10. This is repeated a number of times equal to $\frac{1}{5}$ of the number of versioned objects in the database. This process is itself repeated 15 times, each time for a different versioned matrix chosen uniformly at random.

5. **Insert** - A random modification is made. This is repeated a number of times equal to the number of versioned arrays in the database system, times five. This process is itself repeated 5 times, each time for a different versioned matrix chosen uniformly at random.

The "Head" workload is meant to simulate the versioning system being used as a version-control system, where users generally access the most-recent version but occasionally review past versions. The "Random" workload is meant to represent the system being used to store a collection of read-only matrices that users want to access periodically. The "Range" workload is meant to represent the system being used to store a single growing data set, where users want to view regions within the data set. The "Everything" workload intended to throw the kitchen sink, so to speak, at the system; to stress-test it by throwing a little bit of everything at it.

These workloads were executed on the NOAA weather data set and the ConceptNet data set, in the order listed above. The results are listed in Table 6.14.

Note that the "Compressed" size includes all of the inserted data; for example, the ConceptNet data itself only uses 58MB of disk space when fully compressed.

Prior to the first run, all arrays were delta-compressed in order from newest to oldest. After each workload, a materialization pass was executed: A disk budget was allocated of 10% of the space saved by differencing all of the matrices, as opposed to storing them fully materialized. Matrices were then materialized, per the greedy approach discussed previously and optimizing against past query history, until the budget was exhausted. On subsequent runs, if the materialization pass did not use its 10% budget to materialize an array that had been materialized previously, the array was differenced once again, so as to not exceed the budget.

As can be seen, in general, random reads tend to be considerably more expensive than Head reads. This is because any given random matrix is unlikely to be materialized, so a random read is likely to require reading multiple matrix objects from disk.

Random-range queries also tend to be expensive, but proportionally less so than random reads, despite retrieving more data from the database. Part of the reason for this is that range queries are batchable: In order to retrieve the matrix in the range that is farthest from the relevant materialized matrix, it's necessary to walk through the rest of the range, finding the value of each intermediate matrix. If these values are already being calculated, they might as well be used; so the cost of materializing the most-expensive array in the range is roughly equal to the cost of materializing the entire range. Part of the benefit also comes from the materialization pass: The single-query workload didn't contain enough information to place a materialization usefully, because its queries were focused on the already-materialized head of the versioned matrix. However, after the first random query workload, the available query statistics encouraged the optimizer to materialize matrices in a more distributed pattern, so any given range query was more likely to be close to (or even to contain) a materialized matrix version.

It is worth taking particular note of the ConceptNet data set here, because this is the only experiment run against it. The algorithms discussed herein focus on delta

compression of dense matrices, and the ConceptNet data is sparse. However, it is worth noting that the proposed hybrid differencing algorithm will work properly for sparse data, if not necessarily optimally: Because the data is naturally sparse, the threshold value for the hybrid differencing system will most likely be 0 bits per value. Therefore, the hybrid system will tend to devolve into a simple sparse delta.

It's also worth noting that the ConceptNet workload compressed quite well, by nearly a factor of 100. It's clear from these results that sparse matrices with few differences between version, such as those stored by the ConceptNet data set, can be compressed a great deal. The weather-data set did not compress nearly as well, but it did compress substantially.

Unlike the last round of tests, all of the operations in these tests performed more poorly with compressed data than with uncompressed data, some substantially so. There are several causes for this. First, for the sparse-matrix ConceptNet data, sparse-matrix additions are being performed. The current implementation of sparse addition used by this library is not particularly efficient; it creates a new sparse matrix and copies the values from the two original sparse matrices into the new matrix. For Head queries, recall that this test only flushes the operating system's buffer cache periodically, rather than between each query, to simulate a more-realistic workload where the operating system buffer pool does play some role. For Head lookups in particular, this means that most data will be coming from memory cache, and most of the runtime of the sample query workload is processor overhead. Head matrices should not be differenced against any other matrices, in this implementation, but they can be compressed, and the Lempel-Ziv algorithm does require some number of CPU cycles in order to execute. For uncompressed data, however, if the file in question is already cached in memory, it is simply copied to a new buffer and returned. For the Insert tests, recall that the insertion technique is to read the Head and modify it, which means that Insert will be more expensive with compressed data for the same reason that Head is more expensive.

While these explanations are causes, they are not counterarguments: This test illustrates a use case where the hybrid delta Lempel-Ziv method is considerably slower than an implementation with no compression. However, the performance loss is counterbalanced by a sizable improvement in disk-space requirements. For some applications, this may be a worthwhile tradeoff.

Chapter 7

Future Work

This research has answered a number of questions, but it has raised many questions as well.

Many different possible techniques for compression and data storage have been discussed herein, but due to time and resource constraints, not all of them were investigated. One major uninvestigated area is the storage format of sparse matrices. The storage overhead associated with sparse matrices, particularly high-dimensionality sparse matrices with several integers per coordinate, is quite large with the system used here. It seems likely that a system that is optimized to, for example, operate directly on a sparse matrix that's stored as a run-length-encoded dense matrix; or to be able to store multiple adjacent matrix cells more efficiently, would be much more efficient in some cases.

For dense matrices, these tests did not examine the performance of chunking under a wide variety of scenarios. For example, one interesting astronomy workload might consist of queries to acquire data in the vicinity of a star, in a matrix holding a stored telescope image. It's quite possible that this data would be split across two chunks, in the current scheme. A more-efficient scheme in this case might be to have chunks overlap, such that each chunk stores some overlapping data from all adjacent chunks.

This implementation did not consider caching of queried arrays in an in-memory buffer pool. With delta chains, a simple least-recently-used buffer-pool policy may not be optimal in the general case: The cost of querying for one array is affected by which other arrays exist in the buffer pool. If two arrays are accessed with equal frequency, and both are part of a long common delta chain, better query throughput might be achieved by caching the array that is earlier in the chain, so that the full chain needn't be read from disk if either array is queried.

This arrangement implicitly assumes that arrays will be stored materialized in the buffer pool. As with any compression method, it is possible to cache objects in either their compressed or uncompressed forms. Caching a compressed object increases the time required to retrieve the cached object, since it must be decompressed first. However, caching an uncompressed object uses more memory, and so leaves less space in the buffer pool for other objects. A study of a variety of buffer-pool implementations on top of this system could be of value.

The API used here works with entire matrices, or matrix chunks, in memory at once. This results in a large amount of system memory usage. As has been discussed, the native operating-system memory allocation methods are relatively expensive. Building a custom memory allocator might improve performance substantially. Additionally, an iterator-based approach, that reads data from disk only as needed and uses only a small, constant-sized memory buffer for each step in the process of uncompressing a matrix, would work around the memory-allocation issue altogether and might yield much-better performance.

Lempel-Ziv compression achieves good compression in the general case, but it is a relatively processor-intensive algorithm, and compression algorithms like run-length encoding or bitmap encoding may perform better for certain types of data. It could be interesting to consider means of quickly and efficiently identifying which compression algorithm will likely yield the best result for a given matrix,

without running all compression algorithms on the matrix to view the result.

As was mentioned briefly, the Git version-control system uses a tunable mechanism for finding efficient delta chains of stored objects. Git did perform very slowly, without generating a particularly efficient stored form, with its default settings. However, it could be interesting to explore a variety of Git compression parameters, and to swap out Git's native delta-generation scheme with another more-efficient, matrix-specific scheme.

Alternative methods of matrix serialization, such as Hilbert curves, were considered briefly but not in great depth. It was shown that a simple approach to compression using alternative serialization orders did yield a small decrease in space requirements. A further investigation of a variety of options in this space, particularly regarding higher-dimensionality matrices, might yield interesting results.

These experiments operated under the assumption that the overall performance of matrix retrieval is important. However, this is not always the case. For example, the NOAA stores its weather data as flat files in a file format whose compression is based on the JPEG 2000 standard. As was determined here, JPEG 2000 yields good compression, and therefore small files, but it is very expensive algorithmically. For this data set, that's fine: The data is hosted on a single Web site; the general public can download it. The bottleneck is presumably the one public source; there is only one provider of this data, but arbitrarily many consumers. So in order to optimize around this bottleneck, the data size that the central server needs to work with is minimized at the cost of requiring the many users of the data to each do some additional processing work on their own. It could be of value to reconsider the methods discussed here with an eye towards a central-server model, where disk and network bandwidth are major constraints and CPU cost is less important.

Chapter 8

Conclusion

This research has led to the evaluation of a host of different techniques for storing and compressing sparse and dense matrices, as well as the development of a software toolkit that offers efficient versioned-matrix storage and retrieval.

The aim of this work was to demonstrate a storage mechanism that provides fast access to large matrices and collections of matrices, while using a minimum of disk storage. The techniques proposed herein were able to cut the storage requirements of some real-world collections of matrices by more than a factor of 10, while speeding up certain types of queries by a factor of 2 to 3. This is a clear, substantial improvement.

Based on this work, future matrix database systems such as SciDB will be able to store larger volumes of data more efficiently than their predecessors. Ultimately, this will hopefully be a small step towards helping scientists and researchers transition from a world where each research lab generates their own data, with great thought and expense put into storing and processing it, to a world where smaller data sizes allow faster transmission of data and easier collection and processing of very large data sets.

As much as this work is a conclusion, it is also a starting point. Matrix databases are a young technology, with many areas yet unexplored. While this research offers some specific guidance in the form of a system for efficient matrix storage, it also discusses and provides some benchmarks from a wide variety of techniques and

ideas not used within this proposed system. This set of ideas is presented here with the hope that future developers and researchers can incorporate them into new applications, in particular ideas that adapt matrix storage systems to special cases and special requirements, rather than the nebulous "general case" discussed here. It has been said that no one is exactly average; it is certainly the case that few complex uses of software are exactly typical, or precisely fit how the designers of the software expected it to be used.

The motivations of this work were to develop a practical API and library for versioned matrices, to implement a variety of differencing algorithms, and to consider various broader ideas surrounding differenced storage. The proposed and implemented library fulfills all of these goals, and the tests and ideas that have been discussed provide grounds for further exploring matrix tools.

Bibliography

[1] T. Boutell. PNG (Portable Network Graphics) Specification Version 1.0. RFC 2083, Internet Engineering Task Force, March 1997.

[2] Scott Chacon. *Git Community Book*, chapter 1. the Git SCM community, `http://book.git-scm.com/`, 2010.

[3] Google Chrome development team. Software updates: Courgette. (abstract and summary; full paper pending), 2010.

[4] Victor Eijkhout. Lapack working note 50 distributed sparse data structures for linear algebra operations, 1992.

[5] Francisco Javier Thayer Fbrega, Francisco Javier, and Joshua D. Guttman. Copy on write, 1995.

[6] GraphicsMagick. Graphicsmagick image processing system. `http://www.graphicsmagick.org/`, January 2011.

[7] Various individual contributors. Stock.xchng. `http://www.sxc.hu/`, April 2010.

[8] Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Knapsack Problems*. Springer, Berlin, Germany, 2004.

[9] Yi Lin, Mohan S Kankanhalli, and Tat seng Chua. T.-s.,: Temporal multi-resolution analysis for video segmentation. In *In ACM Multimedia Conference*, pages 494–505, 1999.

[10] Jean loup Gaily <gzip@prep.ai.mit.edu> and Mark Adler <madler@alumni.caltech.edu>. gzip algorithm. `http://www.gzip.org/algorithm.txt`, September 1997.

[11] Paul E. Mckenney, Jonathan Appavoo, Andi Kleen, O. Krieger, Orran Krieger, Rusty Russell, Dipankar Sarma, and Maneesh Soni. Read-copy update. In *In Ottawa Linux Symposium*, pages 338–367, 2001.

[12] Eugene W. Myers. An $o(nd)$ difference algorithm and its variations. *Algorithmica*, 1:251–266, 1986.

[13] Colin Percival. Naive differences of executable code. Unpublished paper, 2003.

[14] R. Rivest. The MD5 Message-Digest Algorithm. RFC 1321, Internet Engineering Task Force, April 1992.

[15] Youcef Saad. Sparskit: a basic tool kit for sparse matrix computations - version 2, 1994.

[16] SciDB. Scidb open source data management and analytics software for scientific research. `http://www.scidb.org/`, December 2010.

[17] Adam Seering, Philippe Cudre-Mauroux, Samuel Madden, and Michael Stonebraker. Efficient versioning for scientific array databases. Paper not yet published, 2011.

[18] Athanassios Skodras, Charilaos Christopoulos, and Touradj Ebrahimi. The jpeg 2000 still image compression standard. *IEEE Signal processing Magazine*, 18:36–58, 2001.

[19] Large Synoptic Survey Telescope. Lsst data management. `http://www.lsst.org/lsst/science/concept_data`, December 2010.

[20] Pauli Virtanen. Weave. `http://www.scipy.org/Weave`, May 2009.

[21] Jeffrey Scott Vitter. Design and analysis of dynamic huffman codes. *Journal of the ACM*, 34:825–845, 1987.

[22] Eric W Weisstein. Hilbert curve. `http://mathworld.wolfram.com/HilbertCurve.html`, March 2010. From MathWorld–A Wolfram Web Resource.

[23] Jian Zhang, Sei-Ichiro Kamata, and Yoshifumi Ueshige. A pseudo-hilbert scan for arbitrarily-sized arrays. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, E90-A(3):682–690, 2007.

[24] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE TRANSACTIONS ON INFORMATION THEORY*, 23(3):337–343, 1977.

[25] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding, 1978.